COVANTEC

COOPERATIVA / VANGUARDISTA / TECNOLÓGICA

Materiales del entrenamiento de programación en Python - Nivel básico

Versión 0.2

Leonardo J. Caballero G.

07 de septiembre de 2022

Covantec R.L., Santa Cruz de Mara, Mara, Zulia. 4046.
Telf. +58-414-979.80.83 / +58-426-771.35.73 / +58-262-879.18.80
Contactos: covantec.ve@outlook.com - https://github.com/Covantec/

Índice general

Repositorio de manuales y recursos del entrenamiento **«Programación en** Python 2.7[1] **- Nivel básico»** realizado por la empresa Covantec R.L[2].

Sobre este entrenamiento

Para dominar el lenguaje de programación se tiene pensado como un entrenamiento de 2 a 3 días para las personas que son nuevas usándolo o los que quieren aprender acerca de las mejores prácticas actuales del desarrollo en Python.

La planificación de este entrenamiento se estima en:

- Un entrenamiento de **nivel básico** (*2 a 3 días*) que cubre los *diez (10) capítulos*.

Tabla de contenidos:

[1] https://docs.python.org/2.7/
[2] https://github.com/Covantec

CAPÍTULO 1

Introducción al lenguaje Python

Python es un lenguaje de programación de propósito general muy poderoso y flexible, a la vez que sencillo y fácil de aprender.

En esta lección se busca introducir al lenguaje Python, sus características, modos de instalación, soporte comunitario, y los recursos mas destacados disponibles en la Web para tomar en cuenta. A continuación el temario de esta lección:

1.1 Acerca de Python

Python es un lenguaje de programación de propósito general muy poderoso y flexible, a la vez que sencillo y fácil de aprender. Es un lenguaje de alto nivel, que permite procesar fácilmente todo tipo de estructuras de datos, tanto numéricos como de texto.

Figura 1.1: Lenguaje de programación Python.

Este lenguaje fue creado a principios de los noventa por Guido van Rossum[3] en los Países Bajos.

Es relativamente joven (Fortran 1957, Pascal 1970, C 1972, Modula-2 1978, Java 1991). Toma características de lenguajes predecesores, incluso, compatibilizando la solución de varios de ellos. Por ejemplo, habilita tres formas de imprimir el valor de una variable: desde el entorno interactivo escribiendo su nombre (como en Basic), usando la función `print`, con concatenación de elementos (al estilo del `write` de Pascal) o bien con patrones de formato (al estilo del `printf` de C).

Es software libre, y está implementado en *todas las plataformas* (página 9) y sistemas operativos habituales.

[3] https://es.wikipedia.org/wiki/Guido_van_Rossum

Figura 1.2: Guido van Rossum en 2006.

1.1.1 Open source

Python se desarrolla bajo una licencia de Open source o código abierto aprobada por OSI, por lo que se puede usar y distribuir libremente, incluso para uso comercial.

Figura 1.3: Logotipo de la Open Source Initiative.

La licencia de Python es administrada por *Python Software Foundation* (página 5).

- Aprenda más sobre la licencia[4].
- Licencia Python en OSI[5].

[4] https://docs.python.org/3/license.html
[5] https://opensource.org/licenses/Python-2.0

- Conozca más sobre la Fundación[6].

Python Software Foundation

La Python Software Foundation (PSF) es una corporación sin fines de lucro 501 (c) (3) que posee los derechos de propiedad intelectual detrás del lenguaje de programación Python. Administramos las licencias de código abierto para Python versión 2.1 y posteriores, y poseemos y protegemos las marcas comerciales asociadas con Python.

Figura 1.4: Python Software Foundation.

También realiza la conferencia PyCon de Norteamérica anualmente, apoyamos otras conferencias de Python en todo el mundo y financiamos el desarrollo relacionado con Python con nuestro programa de subvenciones[7] y financiamos proyectos especiales.

> *La misión de Python Software Foundation es promover, proteger y avanzar el lenguaje de programación Python, y apoyar y facilitar el crecimiento de una comunidad* diversa[8] *e internacional de programadores de Python.*
>
> —De la página de la Declaración de la Misión[9].

Nota: Mayor información consulte https://www.python.org/psf/

1.1.2 Aplicaciones

El Python Package Index (PyPI)[10] o en español significa *Índice de paquetes de Python* alberga miles de módulos de terceros para Python.

Tanto la biblioteca estándar de Python como los módulos aportados por la comunidad permiten infinitas posibilidades.

- Desarrollo web e Internet[11].
- Acceso a la base de datos[12].
- GUIs de escritorio[13].
- Científico y numérico[14].
- Educación[15].
- Programación de red[16].

[6] https://www.python.org/psf-landing/
[7] https://www.python.org/psf/grants/
[8] https://www.python.org/psf/diversity/
[9] https://www.python.org/psf/mission/
[10] https://pypi.org/
[11] https://www.python.org/about/apps/#web-and-internet-development
[12] https://www.python.org/about/apps/#database-access
[13] https://www.python.org/about/apps/#desktop-guis
[14] https://www.python.org/about/apps/#scientific-and-numeric
[15] https://www.python.org/about/apps/#education
[16] https://www.python.org/about/apps/#network-programming

Figura 1.5: Aplicaciones están disponibles en el Python Package Index (PyPI).

- Desarrollo de Software y Juegos[17].

1.1.3 Comunidad

El gran software es soportado por grandes personas. La base de usuarios es entusiasta, dedicada a fomentar el uso del lenguaje y comprometida a que sea diversa y amigable.

Figura 1.6: Comunidad Python reunidad en la PyCon 2018 in Cleveland, Ohio.

Declaración de Diversidad

La *Python Software Foundation* (página 5) y la comunidad a nivel mundial de Python dan la bienvenida y fomentan la participación de todos. La comunidad se basa en el respeto mutuo, la tolerancia y el aliento, y estamos trabajando para ayudarnos mutuamente a cumplir con estos principios. Queremos que nuestra comunidad sea más diversa: sea quien sea, y cualquiera sea su experiencia, le damos la bienvenida.

Nota: Mayor información consulte https://www.python.org/community/diversity/

[17] https://www.python.org/about/apps/#software-development

Listas de correo

Existen listas de correo de Python y grupos de noticias como recursos de la comunidad. Estos recursos están disponibles públicamente de python.org, y son usando como un canal de información o discusión sobre ideas nuevas, e incluso históricamente como otra vía soporte en linea.

Ademas hay la guía de recursos de Python que no están en inglés, que incluye listas de correo, documentación traducida y original que no está en inglés, y otros recursos.

Nota: Mayor información consulte https://www.python.org/community/lists/

Internet Relay Chat - IRC

Existen muchos canales relacionados con Python en la red Internet Relay Chat (IRC) de Freenode. Todos los canales esta disponibles en el servidor de IRC en Freenode[18]. Para conectarse al servidor IRC use *irc.freenode.net* o puede usar la interfaz web de chat IRC de Freenode[19].

Para preguntas cortas, usted puede obtener ayuda inmediata visitando el canal `#python`. Usted necesitará registrar su apodo con FreeNode, usando la guiá de instrucciones para registrar apodo[20].

Nota: Mayor información consulte https://www.python.org/community/irc/

Foros

Existe algunos recursos disponibles en formato de Foros, a continuación se listan:

- Python Forum (English)[21].
- Python-Forum.de (German)[22].
- /r/learnpython (English)[23].

Si usted esta buscando un foro nativo en su idioma, por favor, consulte la pagina de los grupos locales en la Wiki de Python[24].

Comunidades locales

Siendo Python un proyecto *Open source* (página 4) el cual es mantenido por toda una gran comunidad de usuarios y desarrolladores a nivel mundial, la cual ofrece soporte comunitario del proyecto Python en Sudamérica.

Comunidad(es) de Python en Argentina Nuestro objetivo es nuclear a los usuarios de Python. Pretendemos llegar a personas y empresas, promover el uso de Python e intercambiar información. Más información visite http://www.python.org.ar/

Comunidad(es) de Python en Brasil La comunidad Python Brasil reúne grupos de usuarios en todo el Brasil interesados en difundir e divulgar a lenguaje de programación. Más información visite http://python.org.br

Comunidad(es) de Python en Chile Una comunidad de amigos apasionados por la tecnología e informática, que busca compartir conocimiento de forma libre y sin fines de lucro, con el fin de fortalecer a los miembros de la comunidad para generar un impacto positivo en la región. Más información visite https://pythonchile.cl/comunidad/

[18] https://freenode.net/view/Main_Page
[19] https://webchat.freenode.net/
[20] https://old.freenode.net/kb/answer/registration
[21] https://python-forum.io/
[22] https://www.python-forum.de/
[23] https://www.reddit.com/r/learnpython/
[24] https://wiki.python.org/moin/

Comunidad(es) de Python en Colombia Somos una comunidad enfocada en extender en el territorio nacional el uso y aprendizaje de las distintas tecnologías Python y sus numerosas aplicaciones. Más información visite https://www.python.org.co/

Comunidad(es) de Python en Ecuador Somos una comunidad independiente, conformada por entusiastas de Python, con la filosofía común de que el conocimiento debe ser libre. No lucramos con ningún evento, y esperamos llegar a todos aquellos que desean aprender. Más información visite https://python.ec/

Comunidad(es) de Python en Perú Comunidad de estudiantes, profesionales e interesados en tener un espacio donde conversar, preguntar e intercambiar ideas todo acerca del lenguaje de programación Python. Más información visite https://www.meetup.com/es/pythonperu/

Comunidad(es) de Python en Paraguay Las comunidades de Python están presentes en todo el mundo, con el objeto de apoyar a los usuarios de este lenguaje y difundirlo. En Paraguay lo conforman un grupo de entusiastas de Python, que creemos que podemos sumar ganas y esfuerzo para lograr difundir y ayudar a toda aquella persona que se interese por Python a crecer dentro del sus interés. Entre las funciones de la comunidad organizamos eventos de distinto tipo para difundir el lenguaje, desde PyDay, meetup hasta simples reuniones para comentar ideas y experiencias. Más información visite https://pythonpy.org

Comunidad(es) de Python en Uruguay Para más información visite https://python.uy

Comunidad(es) de Python en Venezuela La Fundación Python de Venezuela (FPyVE) es una organización sin fines de lucro creada el 2 de febrero de 2015 dedicada a fomentar al lenguaje de programación Python. El objetivo principal es el desarrollo, ejecución y/o consultoría de proyectos científicos, tecnológicos y productivos, para cumplir fines de interés público en torno al desarrollo Nacional, haciendo uso y promoviendo para tal fin, el uso del lenguaje de programación Python y las tecnologías asociadas a éste. Más información visite http://pyve.github.io/

Ver también:

Consulte la sección de *lecturas suplementarias* (página 261) del entrenamiento para ampliar su conocimiento en esta temática.

1.2 Características

Las características[25] del lenguaje de programación Python se resumen a continuación:

- Es un lenguaje interpretado[26], **no compilado**, usa *tipado dinámico* (página 9), *fuertemente tipado* (página 8).
- Es *multiplataforma* (página 9), lo cual es ventajoso para hacer ejecutable su código fuente entre varios sistema operativos.
- Es un lenguaje de programación multiparadigma[27], el cual soporta varios paradigma de programación como *orientación a objetos* (página 194), *estructurada* (página 99), programación imperativa[28] y, en menor medida, programación funcional[29].
- En Python, el formato del código (p. ej., la indentación) es estructural.

1.2.1 Fuertemente tipado

El fuertemente tipado[30] significa que el tipo de valor no cambia repentinamente. Un *string* (página 46) que contiene solo dígitos no se convierte mágicamente en un número. Cada cambio de tipo requiere una conversión explícita. A continuación un ejemplo de este concepto:

[25] https://es.wikipedia.org/wiki/Python#Características_y_paradigmas
[26] https://es.wikipedia.org/wiki/Lenguaje_interpretado
[27] https://es.wikipedia.org/wiki/Categor%C3%ADa:Lenguajes_de_programaci%C3%B3n_multiparadigma
[28] https://es.wikipedia.org/wiki/Programación_imperativa
[29] https://es.wikipedia.org/wiki/Programación_funcional
[30] https://es.wikipedia.org/wiki/Lenguaje_de_programación_fuertemente_tipado

```
# varible "valor1" guarda un valor entero, varible "valor2" guarda un valor cadena
valor1, valor2 = 2, "5"
# se usa el metodo int() para convertir a entero
total = valor1 + int(valor2)
# se usa el metodo str() para convertir a cadena
print ("El total es: " + str(total))
```

1.2.2 Tipado dinámico

El tipado dinámico[31] significa que los objetos en tiempo de ejecución (valores) tienen un tipo, a diferencia del tipado estático donde las variables tienen un tipo. A continuación un ejemplo de este concepto:

```
# "variable" guarda un valor integer
variable = 11
print (variable, type(variable))

# "variable" guarda un valor string
variable = "activo"
print (variable, type(variable))
```

1.2.3 Multiplataforma

Python es multiplataforma[32], lo cual es ventajoso para hacer ejecutable su código fuente entre varios sistema operativos, eso quiere decir, soporta las siguientes plataformas para su ejecución:

- Versiones Python para Microsoft Windows (y DOS)[33] (arquitectura x86/x86-64 en presentación de ejecutable, archivo Zip, instalador basado en la Web).

  Truco: Para mayor información consulte la sección *Instalando Python en Windows* (página 12).

- Versiones Python para macOS (Macintosh)[34] (arquitectura 32bit/64bit en presentación de instalador ejecutable).

  Truco: Para mayor información consulte la sección *Instalando Python en una Mac* (página 12).

- Versiones Python en código fuente[35] (archivo tarball del código fuente comprimido con XZ y con Gz). Para las mayoría de los sistemas Linux/UNIX, usted debe descargar y compilar el código fuente.

  Truco: Para mayor información consulte la sección *Instalando Python en un Linux* (página 12).

- Versiones de Implementaciones Alternativas Python[36], la versión «tradicional» de Python (tiene nombre código `CPython`). Existen un número de implementaciones alternativas que están disponibles a continuación:
  - IronPython[37], Python ejecutando en .NET.
  - Jython[38], Python ejecutando en el Java Virtual Machine.

[31] https://es.wikipedia.org/wiki/Tipado_dinámico
[32] https://es.wikipedia.org/wiki/Multiplataforma
[33] https://www.python.org/downloads/windows/
[34] https://www.python.org/downloads/macos/
[35] https://www.python.org/downloads/source/
[36] https://www.python.org/download/alternatives/
[37] https://ironpython.net/
[38] https://www.jython.org/

  - PyPy[39], Una rápida implementación de python con un compilador JIT.
  - Stackless Python[40], Una rama del desarrollo del `CPython` que soporta microthreads.
  - MicroPython[41], Python ejecutando en micro controladores.
- Versiones de Python en otras plataformas[42], la versión «tradicional» de Python (tiene nombre código `CPython`), mas esta versión ha sido migrada a un número plataformas especializadas y/o antiguas, a continuación se destacan algunas de ellas.
  - Pythonista[43], Python para iOS, ofrece un completo entorno de desarrollo para escribir scripts Python en su iPad o iPhone.
  - ActivePython[44], Python para Solaris, Usted puede comprarlo (versiones comerciales y comunitarias, incluidos los módulos de computación científica, no de código abierto), o compilar desde una fuente si tiene un compilador de C. Los paquetes UNIX tienen una variedad de versiones de Python para una variedad de versiones de Solaris. Estos utilizan el estándar Sun `pkgadd`.

Nota: Tenga en cuenta que estos migraciones a menudo están muy por detrás de la última versión de Python.

1.2.4 Filosofía «Incluye baterías»

- Python ha mantenido durante mucho tiempo esta filosofía de «baterías incluidas»:

  «Tener una biblioteca estándar rica y versátil que está disponible de inmediato. Sin que el usuario descargue paquetes separados.»
- Esto le da al lenguaje una ventaja en muchos proyectos.
- Las «baterías incluidas» están en la *librería estándar Python* (página 234).

1.2.5 Zen de Python

Es una colección de 20 principios de software que influyen en el diseño del Lenguaje de Programación Python, de los cuales 19 fueron escritos por *Tim Peters* en junio de 1999. El texto es distribuido como dominio público.

El *Zen de Python* está escrito como la entrada informativa número 20 de las propuestas de mejoras de Python (*Python Enhancement Proposals - PEP*), y se puede encontrar en el sitio oficial de Python.

Los principios están listados a continuación:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Lo práctico gana a lo puro.

[39] https://www.pypy.org/
[40] https://github.com/stackless-dev/stackless/wiki/
[41] http://micropython.org/
[42] https://www.python.org/download/other/
[43] http://omz-software.com/pythonista/index.html
[44] https://www.activestate.com/products/python/

- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (`namespaces`) son una gran idea ¡Hagamos más de esas cosas!

También se incluye como un *huevo de pascua*, el cual se puede encontrar, desde el *intérprete de Python* (página 15), ingresar la siguiente sentencia:

```
>>> import this
```

1.3 Ventajas y desventajas

A continuación se presentan algunas ventajas y desventajas que están presentan en el lenguaje Python:

1.3.1 Ventajas

Las ventajas del lenguaje Python son las siguientes:

Simplificado y rápido Este lenguaje simplifica mucho la programación «hace que te adaptes a un modo de lenguaje de programación, Python te propone un patrón». Es un gran lenguaje para scripting, si usted requiere algo rápido (en el sentido de la ejecución del lenguaje), con unas cuantas líneas ya está resuelto.

Elegante y flexible El lenguaje le da muchas herramientas, si usted quiere listas de varios tipo de datos, no hace falta que declares cada tipo de datos. Es un lenguaje tan flexible usted no se preocupa tanto por los detalles.

Programación sana y productiva Programar en Python se convierte en un estilo muy sano de programar: es sencillo de aprender, direccionado a las reglas perfectas, le hace como dependiente de mejorar, cumplir las reglas, el uso de las lineas, de variables». Ademas es un lenguaje que fue hecho con productividad en mente, es decir, Python le hace ser mas productivo, le permite entregar en los tiempos que me requieren.

Ordenado y limpio El orden que mantiene Python, es de lo que más le gusta a sus usuarios, es muy legible, cualquier otro programador lo puede leer y trabajar sobre el programa escrito en Python. Los módulos están bien organizados, a diferencia de otros lenguajes.

Portable Es un lenguaje muy portable (ya sea en Mac, Linux o Windows) en comparación con otros lenguajes. La filosofía de baterías incluidas, son las librerías que más usted necesita al día a día de programación, ya están dentro del interprete, no tiene la necesidad de instalarlas adicionalmente con en otros lenguajes.

Comunidad Algo muy importante para el desarrollo de un lenguaje es la comunidad, la misma comunidad de Python cuida el lenguaje y casi todas las actualizaciones se hacen de manera democrática.

1.3.2 Desventajas

Las desventajas del lenguaje Python son las siguientes:

Curva de aprendizaje La «curva de aprendizaje cuando ya estás en la parte web no es tan sencilla».

Hosting La mayoría de los servidores no tienen soporte a Python, y si lo soportan, la configuración es un poco difícil.

Librerías incluidas Algunas librerías que trae por defecto no son del gusto de amplio de la comunidad, y optan a usar librerías de terceros.

1.4 Instalación

Debido al soporte *multiplataforma* (página 9) de Python, se ofrecen ciertos recursos para los sistemas operativos más populares:

1.4.1 Instalando Python en Windows

- Instalando Python en Windows[45].

1.4.2 Instalando Python en una Mac

- Instalando Python en una Mac[46].

1.4.3 Instalando Python en un Linux

En una distribución estándar Linux dispone por defecto el interprete Python instalado, para comprobar la correcta instalación solamente debería ejecutar el comando en la consola:

```
python
Python 2.7.13 (default, Sep 26 2018, 18:42:22)
[GCC 6.3.0 20170516] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Si le muestra los mensajes anteriores esta correctamente instalado el interprete Python en su Linux.

Si al ejecutar el comando anterior muestra el mensaje:

```
python
bash: python: no se encontró la orden
```

Esto es debido a que no tiene instalado el interprete, así que debe ejecutar el siguiente comando:

```
sudo apt-get install -y python-dev
```

De nuevo vuelva a ejecutar en su consola de comando el comando `python`.

Ver también:

Consulte la sección de *lecturas suplementarias* (página 261) del entrenamiento para ampliar su conocimiento en esta temática.

1.5 Su primer programa

En informática, un programa **Hola Mundo** es el que imprime el texto *«¡Hola, Mundo!»* en un dispositivo de visualización, en la mayoría de los casos una pantalla de monitor. Este programa suele ser usado como introducción al estudio de un lenguaje de programación, siendo un primer ejercicio típico, y se lo considera fundamental desde el punto de vista didáctico.

[45] https://www.youtube.com/watch?v=VTykmP-a2KY
[46] https://es.wikibooks.org/wiki/Python/Instalaci%C3%B3n_de_Python/Python_en_Mac_OS_X

El *Hola Mundo* se caracteriza por su sencillez, especialmente cuando se ejecuta en una interfaz de línea de comandos. En interfaces gráficas la creación de este programa requiere de más pasos.

El programa *Hola Mundo* también puede ser útil como prueba de configuración para asegurar que el compilador, el entorno de desarrollo y el entorno de ejecución estén instalados correctamente y funcionando.

1.5.1 ¡Hola, Mundo!

Programa ¡Hola, Mundo! en diversas versiones de Python:

Python 2.x:

```
print "Hola Mundo"
```

Python 3.x:

```
print("Hola Mundo");
```

1.5.2 Ejecución

Dependiendo del sistema operativo que este usando debe realizar procedimientos distintos para cada plataforma cuando usted quiere escribir y ejecutar un programa Python. A continuación un procedimiento básico para las principales plataformas:

Ejecutar un programa en Windows

Cree un directorio llamado `proyectos` la unidad `C:\` y dentro de este, cree un archivo de texto plano con el siguiente nombre `holamundo.py` y escriba la sintaxis de *Python 2* (página 13) o *Python 3* (página 13) respectivamente.

Luego ejecute desde la consola de `MS-DOS` el siguiente comando:

```
C:\Python27\python C:\proyectos\holamundo.py
```

Usted debe ver la línea *Hola Mundo*.

Enhorabuena, usted ha ejecutado su primer programa Python.

Ejecutar un programa en Mac OSX

1. Haga clic en `Archivo` y luego la nueva Ventana del `Finder`.
2. Haga clic en `Documentos`.
3. Haga clic en `Archivo` y luego en `Nueva carpeta`.
4. Llame a la carpeta `proyectos`.
5. Usted va a almacenar todos los programas relacionados con la clase allí.
6. Haga clic en `Aplicaciones` y, a continuación `TextEdit`.
7. Haga clic en `TextEdit` en la barra de menú y seleccione `Preferencias`.
8. Seleccione `Texto plano`.
9. En el vacío `TextEdit` tipo de ventana en el siguiente programa, tal y como escribe la sintaxis de *Python 2* (página 13) o *Python 3* (página 13) respectivamente.
10. Desde el archivo de menú en TextEdit.
11. Haga clic en `Guardar como`.

12. En el campo Guardar como: escriba `holamundo.py`.
13. Seleccione `Documentos` y la carpeta de archivos `proyectos`.
14. Haga clic en `Guardar`.

Funcionamiento de su Primer Programa

1. Seleccione `Aplicaciones`, a continuación, `Utilidades y Terminal`.
2. En la ventana `Terminal` ejecute `ls` y presione la tecla `Enter`. Se debe dar una lista de todas las carpetas de nivel superior. Usted debe ver la carpeta de `Documentos`.
3. Ejecute `cd Documentos` y presione `Enter`.
4. Ejecute `ls` y presione `Enter` y debería ver la carpeta `proyectos`.
5. Ejecute `cd proyectos` y presione `Enter`.
6. Ejecute `ls` y presione `Enter` y usted debería ver el archivo `holamundo.py`.
7. Para ejecutar el programa, escriba el siguiente comando `python holamundo.py` y presione `Enter`.
8. Usted debe ver la línea *Hola Mundo*.

Enhorabuena, usted ha ejecutado su primer programa Python.

Ejecutar un programa en Linux

Cree un directorio llamado `proyectos` el `home` de su usuario y dentro de este, cree un archivo de texto plano con el siguiente nombre `holamundo.py` y escriba la sintaxis de *Python 2* (página 13) o *Python 3* (página 13) respectivamente.

Luego ejecute desde la consola de comando el siguiente comando:

```
python $HOME/proyectos/holamundo.py
```

Usted debe ver la línea *Hola Mundo*.

Enhorabuena, usted ha ejecutado su primer programa Python.

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `holamundo.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python holamundo.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

CAPÍTULO 2

Introspección del lenguaje Python

En Python siendo un lenguaje interpretado tiene a disposición un shell de comando para hacer introspección del lenguaje, ademas también existe una forma de hacer mas interactiva la introspección del lenguaje, usando el paquete `ipython`.

En esta lección se busca introducir a la introspección del lenguaje Python usando el interprete como el modo interactivo del paquete adicional `ipython`, y las ventajas aplicar la técnica de **introspección** en sus practicas de programación diarias con el lenguaje. A continuación el temario de esta lección:

2.1 Inmersión al modo interactivo

La *inmersión al modo interactivo* le permite a cualquier usuario el cual **NUNCA** ha trabajando con el interprete de Python[47] pueda tener un primer acercamiento **SIN PROGRAMAR**, solamente con conocer el uso del interprete y sus comandos básicos usando la técnica de introspección.

2.1.1 Introspección en Python

En Python como usted lo ira entendiendo **todo en Python es un objeto**, y la técnica de introspección, no es más que código el cual examina como objetos otros módulos y funciones en memoria, obtiene información sobre ellos y los que los maneja.

De paso, usted podrá definir las funciones sin nombre, las llamará a funciones con argumentos sin orden, y podrá hacer referencia a funciones cuyos nombres desconocemos.

2.1.2 Python a través de su interprete

Es importante conocer Python a través de su interprete debido a varios factores:

- Conocer las clases, sus funciones y atributos propios, a través de la introspección del lenguaje.
- Disponibilidad de consultar la documentación del lenguaje desde el interprete, por mucho tiempo no estaba disponible documentación tipo Javadoc[48] o diagramas de clases[49] del propio lenguaje por lo cual muchas

[47] https://www.python.org/
[48] https://es.wikipedia.org/wiki/Javadoc
[49] https://es.wikipedia.org/wiki/Diagrama_de_clases

programadores **Python** se acostumbraron a estudiar su código de esta forma, así que le recomiendo que use el interprete `python` para eso.

- Hoy en día existente herramientas que te permiten generar documentación desde los códigos fuentes Python como Sphinx[50].

La forma mas fácil es iniciar tu relación con Python simplemente ejecutando el comando `python` de la siguiente forma:

```
python
Python 2.7.13 (default, Sep 26 2018, 18:42:22)
[GCC 6.3.0 20170516] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Puede solicitar la ayudar del interprete de Python, ejecutando:

```
>>> help
Type help() for interactive help, or help(object) for help about object.
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".
```

Para ejecutar la ayuda disponible sobre la sintaxis Python ejecute el siguiente comando:

```
help> modules

Please wait a moment while I gather a list of all available modules...

BaseHTTPServer      asynchat            imputil             sha
Bastion             asyncore            inspect             shelve
CDROM               atexit              io                  shlex
CGIHTTPServer       audiodev            ipython_genutils    shutil
Canvas              audioop             itertools           shutil_backports
ConfigParser        autoreload          jinja2              signal
Cookie              babel               json                simplegeneric
DLFCN               backports           keyword             site
Dialog              base64              lib2to3             sitecustomize
DocXMLRPCServer     bdb                 linecache           six
FileDialog          binascii            linuxaudiodev       smtpd
FixTk               binhex              locale              smtplib
HTMLParser          bisect              logging             sndhdr
IN                  bsddb               macpath             snowballstemmer
IPython             bz2                 macurl2path         socket
MimeWriter          cPickle             mailbox             sphinx
Queue               cProfile            mailcap             sphinx_rtd_theme
ScrolledText        cStringIO           markupbase          spwd
SimpleDialog        calendar            markupsafe          sqlite3
SimpleHTTPServer    cgi                 marshal             sre
```

(continué en la próxima página)

[50] https://en.wikipedia.org/wiki/Sphinx_%28documentation_generator%29

(proviene de la página anterior)

```
SimpleXMLRPCServer  cgitb               math                sre_compile
SocketServer        chunk               md5                 sre_constants
StringIO            cmath               mhlib               sre_parse
TYPES               cmd                 mimetools           ssl
Tix                 code                mimetypes           stat
Tkconstants         codecs              mimify              statvfs
Tkdnd               codeop              mmap                storemagic
Tkinter             collections         modulefinder        string
UserDict            colorsys            multifile           stringold
UserList            commands            multiprocessing     stringprep
UserString          compileall          mutex               strop
_LWPCookieJar       compiler            netrc               struct
_MozillaCookieJar   contextlib          new                 subprocess
__builtin__         cookielib           nis                 sunau
__future__          copy                nntplib             sunaudio
_abcoll             copy_reg            ntpath              symbol
_ast                crypt               nturl2path          sympyprinting
_bisect             csv                 numbers             symtable
_bsddb              ctypes              opcode              sys
_codecs             curses              operator            sysconfig
_codecs_cn          cythonmagic         optparse            syslog
_codecs_hk          datetime            os                  tabnanny
_codecs_iso2022     dbhash              os2emxpath          tarfile
_codecs_jp          dbm                 ossaudiodev         telnetlib
_codecs_kr          decimal             parser              tempfile
_codecs_tw          decorator           pathlib2            termios
_collections        difflib             pdb                 test
_csv                dircache            pexpect             tests
_ctypes             dis                 pickle              textwrap
_ctypes_test        distutils           pickleshare         this
_curses             doctest             pickletools         thread
_curses_panel       docutils            pip                 threading
_elementtree        dumbdbm             pipes               time
_functools          dummy_thread        pkg_resources       timeit
_hashlib            dummy_threading     pkgutil             tkColorChooser
_heapq              easy_install        platform            tkCommonDialog
_hotshot            email               plistlib            tkFileDialog
_io                 encodings           popen2              tkFont
_json               ensurepip           poplib              tkMessageBox
_locale             enum                posix               tkSimpleDialog
_lsprof             errno               posixfile           toaiff
_md5                exceptions          posixpath           token
_multibytecodec     fcntl               pprint              tokenize
_multiprocessing    filecmp             profile             trace
_osx_support        fileinput           prompt_toolkit      traceback
_pyio               fnmatch             pstats              traitlets
_random             formatter           pty                 ttk
_scandir            fpectl              ptyprocess          tty
_sha                fpformat            pwd                 turtle
_sha256             fractions           py_compile          types
_sha512             ftplib              pyclbr              unicodedata
_socket             functools           pydoc               unittest
_sqlite3            future_builtins     pydoc_data          urllib
_sre                gc                  pyexpat             urllib2
_ssl                gdbm                pygments            urlparse
_strptime           genericpath         pytz                user
_struct             getopt              quopri              uu
_symtable           getpass             random              uuid
_sysconfigdata      gettext             re                  warnings
_sysconfigdata_nd   glob                readline            wave
_testcapi           grp                 repr                wcwidth
```

(continué en la próxima página)

(proviene de la página anterior)

```
_threading_local    gzip                resource            weakref
_tkinter            hashlib             rexec               webbrowser
_warnings           heapq               rfc822              wheel
_weakref            hmac                rlcompleter         whichdb
_weakrefset         hotshot             rmagic              wsgiref
abc                 htmlentitydefs      robotparser         xdrlib
aifc                htmllib             runpy               xml
alabaster           httplib             scandir             xmllib
antigravity         ihooks              sched               xmlrpclib
anydbm              imaplib             select              xxsubtype
argparse            imghdr              sets                zipfile
array               imp                 setuptools          zipimport
ast                 importlib           sgmllib             zlib

Enter any module name to get more help.  Or, type "modules spam" to search
for modules whose descriptions contain the word "spam".
```

Entonces consulte la ayuda del módulo `os`, ejecutando:

```
help> os
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

FILE
    /usr/lib/python2.7/os.py

MODULE DOCS
    https://docs.python.org/library/os

DESCRIPTION
    This exports:
      - all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc.
      - os.path is one of the modules posixpath, or ntpath
      - os.name is 'posix', 'nt', 'os2', 'ce' or 'riscos'
      - os.curdir is a string representing the current directory ('.' or ':')
      - os.pardir is a string representing the parent directory ('..' or '::')
      - os.sep is the (or a most common) pathname separator ('/' or ':' or '\\')
      - os.extsep is the extension separator ('.' or '/')
      - os.altsep is the alternate pathname separator (None or '/')
      - os.pathsep is the component separator used in $PATH etc
      - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
      - os.defpath is the default search path for executables
      - os.devnull is the file path of the null device ('/dev/null', etc.)

    Programs that import and use 'os' stand a better chance of being
    portable between different platforms.  Of course, they must then
    only use functions that are defined by all platforms (e.g., unlink
    and opendir), and leave all pathname manipulation to os.path
    (e.g., split and join).
:
```

Truco: Presione la tecla `q` para salir de la ayuda del módulo `os`.

Seguidamente presione la combinación de tecla **Crtl+d** para salir de la ayuda.

Luego realice la importación de la librería del estándar[51] Python llamada `os`, con el siguiente comando:

[51] https://docs.python.org/2/library/index.html

```
>>> import os
>>>
```

Previamente importada la librería usted puede usar la función `dir()` para listar o descubrir que atributos, métodos de la clase están disponibles con la importación

```
>>> dir(os)
['EX_CANTCREAT', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR', 'EX_NOHOST',
'EX_NOINPUT', 'EX_NOPERM', 'EX_NOUSER', 'EX_OK', 'EX_OSERR', 'EX_OSFILE',
'EX_PROTOCOL', 'EX_SOFTWARE', 'EX_TEMPFAIL', 'EX_UNAVAILABLE',
'EX_USAGE', 'F_OK', 'NGROUPS_MAX', 'O_APPEND', 'O_CREAT', 'O_DIRECT',
'O_DIRECTORY', 'O_DSYNC', 'O_EXCL', 'O_LARGEFILE', 'O_NDELAY',
'O_NOCTTY', 'O_NOFOLLOW', 'O_NONBLOCK', 'O_RDONLY', 'O_RDWR', 'O_RSYNC',
'O_SYNC', 'O_TRUNC', 'O_WRONLY', 'P_NOWAIT', 'P_NOWAITO', 'P_WAIT',
'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'TMP_MAX', 'UserDict',
'WCONTINUED', 'WCOREDUMP', 'WEXITSTATUS', 'WIFCONTINUED', 'WIFEXITED',
'WIFSIGNALED', 'WIFSTOPPED', 'WNOHANG', 'WSTOPSIG', 'WTERMSIG',
'WUNTRACED', 'W_OK', 'X_OK', '_Environ', '__all__', '__builtins__',
'__doc__', '__file__', '__name__', '_copy_reg', '_execvpe', '_exists',
'_exit', '_get_exports_list', '_make_stat_result',
'_make_statvfs_result', '_pickle_stat_result', '_pickle_statvfs_result',
'_spawnvef', 'abort', 'access', 'altsep', 'chdir', 'chmod', 'chown',
'chroot', 'close', 'confstr', 'confstr_names', 'ctermid', 'curdir',
'defpath', 'devnull', 'dup', 'dup2', 'environ', 'errno', 'error',
'execl', 'execle', 'execlp', 'execlpe', 'execv', 'execve', 'execvp',
'execvpe', 'extsep', 'fchdir', 'fdatasync', 'fdopen', 'fork', 'forkpty',
'fpathconf', 'fstat', 'fstatvfs', 'fsync', 'ftruncate', 'getcwd',
'getcwdu', 'getegid', 'getenv', 'geteuid', 'getgid', 'getgroups',
'getloadavg', 'getlogin', 'getpgid', 'getpgrp', 'getpid', 'getppid',
'getsid', 'getuid', 'isatty', 'kill', 'killpg', 'lchown', 'linesep',
'link', 'listdir', 'lseek', 'lstat', 'major', 'makedev', 'makedirs',
'minor', 'mkdir', 'mkfifo', 'mknod', 'name', 'nice', 'open', 'openpty',
'pardir', 'path', 'pathconf', 'pathconf_names', 'pathsep', 'pipe',
'popen', 'popen2', 'popen3', 'popen4', 'putenv', 'read', 'readlink',
'remove', 'removedirs', 'rename', 'renames', 'rmdir', 'sep', 'setegid',
'seteuid', 'setgid', 'setgroups', 'setpgid', 'setpgrp', 'setregid',
'setreuid', 'setsid', 'setuid', 'spawnl', 'spawnle', 'spawnlp',
'spawnlpe', 'spawnv', 'spawnve', 'spawnvp', 'spawnvpe', 'stat',
'stat_float_times', 'stat_result', 'statvfs', 'statvfs_result',
'strerror', 'symlink', 'sys', 'sysconf', 'sysconf_names', 'system',
'tcgetpgrp', 'tcsetpgrp', 'tempnam', 'times', 'tmpfile', 'tmpnam',
'ttyname', 'umask', 'uname', 'unlink', 'unsetenv', 'urandom', 'utime',
'wait', 'wait3', 'wait4', 'waitpid', 'walk', 'write']
>>>
```

Otro ejemplo de uso, es poder usar el método `file` para determinar la ubicación de la librería importada de la siguiente forma:

```
>>> os.__file__
'/usr/lib/python2.7/os.pyc'
>>>
```

También puede consultar la documentación de la librería `os` ejecutando el siguiente comando:

```
>>> print os.__doc__
OS routines for NT or Posix depending on what system we're on.

This exports:
  - all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc.
  - os.path is one of the modules posixpath, or ntpath
  - os.name is 'posix', 'nt', 'os2', 'ce' or 'riscos'
  - os.curdir is a string representing the current directory ('.' or ':')
```

(continué en la próxima página)

(proviene de la página anterior)

```
  - os.pardir is a string representing the parent directory ('..' or '::')
  - os.sep is the (or a most common) pathname separator ('/' or ':' or '\\')
  - os.extsep is the extension separator ('.' or '/')
  - os.altsep is the alternate pathname separator (None or '/')
  - os.pathsep is the component separator used in $PATH etc
  - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
  - os.defpath is the default search path for executables
  - os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms.  Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).
>>>
```

Ejecute el comando exit() para salir del interprete...

```
>>> exit()
```

2.1.3 Interprete ipython

Para mejorar la experiencia con el interprete Python le sugerimos instalar el paquete `ipython`, según su documentación:

Según Wikipedia

> «`ipython` es un shell interactivo que añade funcionalidades extra al modo interactivo[52] incluido con Python, como resaltado de líneas y errores mediante colores, una sintaxis adicional para el shell, completado automático mediante tabulador de variables, módulos y atributos; entre otras funcionalidades. Es un componente del paquete SciPy[53].»

Para mayor información visite su página principal de ipython[54] y si necesita instalar este programa ejecute el siguiente comando:

```
sudo apt-get install ipython
```

Luego cierra sesión de **root** y vuelve al usuario y sustituya el comando `python` por `ipython` de la siguiente forma:

```
ipython
Python 2.7.13 (default, Sep 26 2018, 18:42:22)
Type "copyright", "credits" or "license" for more information.

IPython 5.8.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

    In [1]:
```

Un ejemplo de uso del comando `help` es consultar la ayuda del comando `dir` y se ejecuta de la siguiente forma:

```
In [1]: help(dir)
Help on built-in function dir in module __builtin__:
```

(continué en la próxima página)

[52] https://es.wikipedia.org/wiki/Python#Modo_interactivo
[53] https://en.wikipedia.org/wiki/SciPy
[54] https://ipython.readthedocs.io/

(proviene de la página anterior)

```
dir(...)
    dir([object]) -> list of strings

    Return an alphabetized list of names comprising (some of) the
    attributes of the given object, and of attributes reachable
    from it:

    No argument:  the names in the current scope.
    Module object:  the module attributes.
    Type or class object:  its attributes, and recursively the
    attributes of its bases.
    Otherwise:  its attributes, its class's attributes, and
    recursively the attributes of its class's base classes.
```

Entonces presione la tecla **q** para salir de la ayuda de la función `dir()`.

De nuevo realice la importación de la librería del estándar Python llamada `os`.

```
In [2]: import os
```

También consultar los detalles acerca del "objeto" para esto use como ejemplo la librería `os` ejecutando el siguiente comando:

```
In [2]: os?
Type:        module
String form: <module 'os' from '/usr/lib/python2.7/os.pyc'>
File:        /usr/lib/python2.7/os.py
Docstring:
OS routines for NT or Posix depending on what system we're on.

This exports:
  - all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc.
  - os.path is one of the modules posixpath, or ntpath
  - os.name is 'posix', 'nt', 'os2', 'ce' or 'riscos'
  - os.curdir is a string representing the current directory ('.' or ':')
  - os.pardir is a string representing the parent directory ('..' or '::')
  - os.sep is the (or a most common) pathname separator ('/' or ':' or '\\')
  - os.extsep is the extension separator ('.' or '/')
  - os.altsep is the alternate pathname separator (None or '/')
  - os.pathsep is the component separator used in $PATH etc
  - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
  - os.defpath is the default search path for executables
  - os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms.  Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).
```

Escriba la librería *os.* y luego escribe dos **underscore** y presione *dos veces la tecla tabular* para usar la completado automático del interprete al estilo de completación de lineas de comandos[55] en el shell UNIX/Linux para ayudar a la introspección del lenguaje y sus librerías.

```
In [3]: os.__
os.__all__      os.__file__
os.__builtins__ os.__name__
os.__doc__      os.__package__
```

[55] https://en.wikipedia.org/wiki/Command_line_completion

De nuevo ejecute el método `file` para determinar la ubicación de la librería importada

```
In [4]: os.__file__
Out[4]: '/usr/lib/python2.7/os.pyc'
```

También puede consultar la documentación de la librería `os` de la siguiente forma:

```
In [5]: print os.__doc__
OS routines for NT or Posix depending on what system we're on.

This exports:
  - all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc.
  - os.path is one of the modules posixpath, or ntpath
  - os.name is 'posix', 'nt', 'os2', 'ce' or 'riscos'
  - os.curdir is a string representing the current directory ('.' or ':')
  - os.pardir is a string representing the parent directory ('..' or '::')
  - os.sep is the (or a most common) pathname separator ('/' or ':' or '\\')
  - os.extsep is the extension separator ('.' or '/')
  - os.altsep is the alternate pathname separator (None or '/')
  - os.pathsep is the component separator used in $PATH etc
  - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
  - os.defpath is the default search path for executables
  - os.devnull is the file path of the null device ('/dev/null', etc.)

Programs that import and use 'os' stand a better chance of being
portable between different platforms.  Of course, they must then
only use functions that are defined by all platforms (e.g., unlink
and opendir), and leave all pathname manipulation to os.path
(e.g., split and join).
```

Otro ejemplo es imprimir el **nombre de la clase** con el siguiente comando:

```
In [6]: os.__name__
Out[6]: 'os'
```

Y otra forma de consultar la documentación de la librería `os` es ejecutando el siguiente comando:

```
In [7]: help(os)
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

FILE
    /usr/lib/python2.7/os.py

MODULE DOCS
    https://docs.python.org/library/os

DESCRIPTION
    This exports:
      - all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc.
      - os.path is one of the modules posixpath, or ntpath
      - os.name is 'posix', 'nt', 'os2', 'ce' or 'riscos'
      - os.curdir is a string representing the current directory ('.' or ':')
      - os.pardir is a string representing the parent directory ('..' or '::')
      - os.sep is the (or a most common) pathname separator ('/' or ':' or '\\')
      - os.extsep is the extension separator ('.' or '/')
      - os.altsep is the alternate pathname separator (None or '/')
      - os.pathsep is the component separator used in $PATH etc
      - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
      - os.defpath is the default search path for executables
      - os.devnull is the file path of the null device ('/dev/null', etc.)
```

(continué en la próxima página)

(proviene de la página anterior)

```
    Programs that import and use 'os' stand a better chance of being
    portable between different platforms.  Of course, they must then
    only use functions that are defined by all platforms (e.g., unlink
    and opendir), and leave all pathname manipulation to os.path
    (e.g., split and join).
:
```

Entonces presione la tecla `q` para salir de la ayuda del módulo `os`.

Y para cerrar la sesión con el `ipython` ejecute el siguiente comando:

```
In [8]: exit()
Do you really want to exit ([y]/n)? y
```

2.1.4 Interprete bpython

Alternativamente puedes usar el paquete *bpython* que mejora aun mas la experiencia de trabajo con el paquete *ipython*.

Para mayor información visite su página principal de interprete bpython[56] y si necesita instalar este programa ejecute el siguiente comando:

```
sudo apt-get install python-pip
sudo pip install bpython
```

Luego cierra sesión de **root** y vuelve al usuario y sustituya el comando `python` por `ipython` de la siguiente forma:

```
bpython
```

Dentro de interprete Python puede apreciar que ofrece otra forma de presentar la documentación y la estructura del lenguaje, con los siguientes comandos de ejemplos:

```
>>> print 'Hola Mundo'
Hola Mundo
>>> for item in xrange(
+-------------------------------------------------------------------------------+
| xrange: ([start, ] stop[, step])                                              |
| xrange([start,] stop[, step]) -> xrange object                                |
|                                                                               |
| Like range(), but instead of returning a list, returns an object that         |
| generates the numbers in the range on demand.  For looping, this is           |
| slightly faster than range() and more memory efficient.                       |
+-------------------------------------------------------------------------------+

 <C-r> Rewind  <C-s> Save  <F8> Pastebin  <F9> Pager  <F2> Show Source
```

2.1.5 Conclusiones

Como puede apreciar este tutorial no le enseña a programar sino a simplemente aprender a conocer como manejarse en shell de Python y en el modo interactivo usando el paquete `ipython` y otros adicionales como `bpython`, con el fin de conocer a través de la introspección del lenguaje, las librerías estándar y módulos propios escritos en Python que tienes instalado en tu sistema.

Ver también:

[56] https://bpython-interpreter.org/

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

CAPÍTULO 3

Tipos y estructuras de datos

En Python tiene varios tipos de datos *compuestos* estándar disponibles por defecto en el interprete, como los tipos *numéricos*, *secuencias*, *mapeos* y *conjuntos* usados para agrupar otros valores.

Para el caso de las estructuras de datos se usan variables y constantes las cuales usan operadores para tratar los tipos de datos estándar.

En esta lección se describen las variables, operadores y sus tipos de datos en el lenguaje Python, los cuales se resumieron en esta tabla. A continuación el temario de esta lección:

3.1 Jerarquía de tipos estándar

A continuación se muestra una lista de los tipos que están integrados en Python. Los módulos de extensión (escritos en C, Java u otros lenguajes, dependiendo de la implementación) pueden definir tipos adicionales. Las versiones futuras de Python pueden agregar tipos a la jerarquía de tipos (por ejemplo, números racionales, arrays de enteros almacenados eficientemente, etc.).

Algunas de las descripciones de tipo a continuación contienen un párrafo que enumera los «atributos especiales». Estos son atributos que proporcionan acceso a la implementación y no están destinados para uso general. Su definición puede cambiar en el futuro.

En Python tiene varios tipos de datos *compuestos* estándar disponibles por defecto en el interprete, como los tipos *numéricos*, *secuencias*, *mapeos* y *conjuntos* usados para agrupar otros valores.

Para el caso de las estructuras de datos se usan variables y constantes las cuales usan operadores para tratar los tipos de datos estándar.

3.1.1 Clasificación

Los tipos de datos *compuestos* estándar se pueden clasificar como los dos siguientes:

- **Mutable:** su contenido (o dicho valor) puede cambiarse en tiempo de ejecución.
- **Inmutable:** su contenido (o dicho valor) no puede cambiarse en tiempo de ejecución.

Se pueden resumir los tipos de datos *compuestos* estándar en la siguiente tabla:

Categoría de tipo	Nombre	Descripción
Números inmutables	`int`	*entero* (página 41)
	`long`	*entero long* (página 41)
	`float`	*coma flotante* (página 42)
	`complex`	*complejo* (página 43)
	`bool`	*booleano* (página 44)
Secuencias inmutables	`str`	*cadena de caracteres* (página 46)
	`unicode`	*cadena de caracteres Unicode* (página 47)
	`tuple`	*tupla* (página 63)
	`xrange`	*rango inmutable* (página 121)
Secuencias mutables	`list`	*lista* (página 58)
	`range`	*rango mutable* (página 120)
Mapeos	`dict`	*diccionario* (página 65)
Conjuntos mutables	`set`	*conjunto mutable* (página 75)
Conjuntos inmutables	`frozenset`	*conjunto inmutable* (página 75)

Otros tipos de datos incorporados, se describen a continuación:

Categoría de tipo	Nombre	Descripción
Objeto integrado	`NoneType`	el objeto *None* (página 210).
Objeto integrado	`NotImplementedType`	el objeto *NotImplemented* (página 210).
Objeto integrado	`ellipsis`	el objeto *Ellipsis* (página 210).
Objeto integrado	`file`	el objeto *file* (página 214).

3.1.2 Objectos Type

Los objectos Type representan los diversos tipos de objetos. Un objecto type es accedido por la función integrada *type()* (página 122). No hay operaciones especiales en los tipos. El módulo estándar `types` defines los nombres para todos los tipos integrados estándar.

Los tipos son escritos como esto: «<type “int”>».

3.2 Variables y constantes

3.2.1 Variables

Es un nombre que se refiere a un objeto que reside en la memoria. El objeto puede ser de alguno de los tipos vistos (número o cadena de texto), o alguno de los otros tipos existentes en Python.

Cada variable debe tener un nombre único llamado identificador. Eso es muy de ayuda pensar las variables como contenedores que contienen data el cual puede ser cambiado después a través de técnicas de programación.

Alcance de las variables

Las variables en Python son locales por defecto. Esto quiere decir que las variables definidas y utilizadas en el bloque de código de una *función* (página 100), sólo tienen existencia dentro de la misma, y no interfieren con otras variables del resto del código.

A su vez, las variables existentes fuera de una *función* (página 100), no son visibles dentro de la misma.

En caso de que sea conveniente o necesario, una variable local puede convertirse en una variable global declarándola explícitamente como tal con la sentencia *global* (página 31).

Ejemplos de variables

A continuación, se presentan algunos ejemplos del uso de *variables*:

Ejemplo de asignar valor a variable

A continuación, se creará un par de variables a modo de ejemplo. Una de tipo *cadenas de caracteres* (página 46) y una de tipo *entero* (página 41):

```
>>> c = "Hola Mundo" # cadenas de caracteres
>>> type(c) # comprobar tipo de dato
<type 'str'>
>>> e = 23 # número entero
>>> type(e) # comprobar tipo de dato
<type 'int'>
```

Como puede ver en Python, a diferencia de muchos otros lenguajes, no se declara el tipo de la variable al crearla. En *Java*, por ejemplo, definir una variable seria así:

```
String c = "Hola Mundo";
int e = 23;
```

También nos ha servido el pequeño ejemplo para presentar los comentarios en linea en Python: cadenas de caracteres que comienzan con el carácter # y que Python ignora totalmente. Hay más tipos de *comentarios* (página 49), de los cuales se tratarán más adelante.

Ejemplo de cambiar valor a variable

A continuación, se cambiará el valor para una variable de tipo *cadenas de caracteres* (página 46) a modo de ejemplo:

```
>>> c = "Hola Plone" # cadenas de caracteres
>>> c
'Hola Plone'
```

Ejemplo de asignar múltiples valores a a múltiples variables

A continuación, se creará múltiples variables (*entero* (página 41), *coma flotante* (página 42), *cadenas de caracteres* (página 46)) asignando múltiples valores:

```
>>> a, b, c = 5, 3.2, "Hola"
>>> print a
5
>>> print b
3.2
>>> print c
'Hola'
```

Si usted quiere asignar el mismo valor a múltiples variables al mismo tiempo, usted puede hacer lo siguiente:

```
>>> x = y = z = True
>>> print x
True
>>> print y
True
>>> print z
True
```

El segundo programa asigna el mismo valor booleano a todas las tres variables `x`, `y`, `z`.

3.2.2 Constantes

Una constante es un tipo de variable la cual no puede ser cambiada. Eso es muy de ayuda pensar las constantes como contenedores que contienen información el cual no puede ser cambiado después.

En Python, las constantes son usualmente declaradas y asignadas en un módulo. Aquí, el módulo significa un nuevo archivo que contiene variables, funciones, etc; el cual es importada en el archivo principal. Dentro del módulo, las constantes son escritas en letras MAYÚSCULAS y separadas las palabras con el carácter *underscore* _.

Constantes integradas

Un pequeño número de constantes vive en el espacio de nombres incorporado. Son las siguientes:

None Más información consulte sobre *None* (página 210).

NotImplemented Más información consulte sobre *NotImplemented* (página 210).

Ellipsis Más información consulte sobre *Ellipsis* (página 210).

False El valor falso del tipo *booleano* (página 44).

True El valor verdadero del tipo *booleano* (página 44).

__debug__ Esta constante su valor es `True` si Python no se inició con una opción `-O`. Véase también la sentencia *assert* (página 186).

Nota: Los nombres *None* (página 210) y `__debug__` no se pueden reasignar (asignaciones a ellos, incluso como un nombre de atributo, causa una excepción *SyntaxError* (página 192)), por lo que pueden considerarse constantes «verdaderas».

Ejemplo de constantes

A continuación, se presentan algunos ejemplos del uso de *constantes*:

Ejemplo de constantes desde un módulo externo

Crear un archivo llamado `constantes.py` con el siguiente contenido:

```
IP_DB_SERVER = "127.0.0.1"
PORT_DB_SERVER = 3307
USER_DB_SERVER = "root"
PASSWORD_DB_SERVER = "123456"
DB_NAME = "nomina"
```

Crear un archivo llamado `main.py` con el siguiente contenido:

```
import constantes

print ("scp -v -P {0} {1}@{2}:/{3}/{4}/{4}.sql /srv/backup".format(
    str(constantes.PORT_DB_SERVER), constantes.USER_DB_SERVER,
    constantes.IP_DB_SERVER, constantes.USER_DB_SERVER,
    constantes.DB_NAME))
```

Luego ejecuta el programa de la siguiente forma:

```
python main.py
```

Cuando usted ejecuta el programa, la salida será:

```
scp -v -P 3307 root@127.0.0.1:/root/webapp/db.sql /srv/backup
```

En el programa anterior, existe un archivo de módulo `constantes.py`. Entonces en este se asignan los valores de constantes `IP_DB_SERVER`, `PORT_DB_SERVER`, `USER_DB_SERVER`, `PASSWORD_DB_SERVER` y `DB_NAME`. Ademas, existe el archivo de módulo `main.py` el cual importa el módulo `constantes`. Finalmente, se imprime una linea de conexión del comando `scp` de Linux usando la función integrada en la librería estándar Python llamada *format()* (página 131).

Nota: En realidad, no se usa las constantes en Python. El módulo `globals` o `constants` es usado a lo largo de los programas de Python.

3.2.3 Palabras reservadas

Existen ciertas palabras que tienen significado especial para el intérprete de Python. Estas no pueden utilizarse para ningún otro fin (como ser nombrar valores) excepto para el que han sido creadas. Estas son:

- *and* (página 86).
- `as`.
- *assert* (página 186).
- *break* (página 89).
- *class* (página 200).
- *continue* (página 89).
- *def* (página 100).
- *del* (página 31).
- *elif* (página 84).
- *else* (página 84).
- *except* (página 184).
- `exec`.
- *finally* (página 188).
- *for* (página 90).
- *from* (página 162).
- *global* (página 31).
- *if* (página 84).
- *import* (página 159).
- *in* (página 84).
- *is* (página 84).
- *lambda* (página 112).
- *not* (página 86).
- *or* (página 86).
- *pass* (página 104).
- *print* (página 153).
- *raise* (página 186).

- *return* (página 104).
- *try* (página 184).
- *while* (página 88).
- *with* (página 189).
- `yield`.

Nota: Para Python 2.7 son un total de 31 palabras reservadas.

Puede verificar si una palabra esta reservada utilizando el módulo integrado `keyword`, de la siguiente forma:

```
>>> import keyword
>>> keyword.iskeyword('as')
True
>>> keyword.iskeyword('x')
False
```

Para obtener una lista de todas las palabras reservadas

```
>>> import keyword
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def',
'del', 'elif', 'else', 'except', 'exec', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not',
'or', 'pass', 'print', 'raise', 'return', 'try', 'while',
'with', 'yield']
```

3.2.4 Reglas y convención de nombres

Algunas reglas y convenciones de nombres para las *variables* (página 26) y *constantes* (página 28):

- Nunca use símbolos especiales como !, @, #, $, %, etc.
- El primer carácter no puede ser un número o dígito.
- Las constantes son colocadas dentro de módulos Python y significa que no puede ser cambiado.
- Los nombres de constante y variable debería tener la combinación de letras en minúsculas (de *a* a la *z*) o MAYÚSCULAS (de la *A* a la *Z*) o dígitos (del *0* al *9*) o un `underscore` (_). Por ejemplo:
  - snake_case
  - MACRO_CASE
  - camelCase
  - CapWords
- Los nombres que comienzan con guión bajo (simple _ o doble ___) se reservan para variables con significado especial
- No pueden usarse como identificadores, las *palabras reservadas* (página 29) .

3.2.5 Sentencia del

La sentencia `del` se define recursivamente muy similar a la forma en el cual se define la asignación. A continuación unos ejemplos donde se inicializan variables:

```
>>> cadena, numero, lista = "Hola Plone", 123456, [7,8,9,0]
>>> tupla = (11, "Chao Plone", True, None)
>>> diccionario = {"nombre":"Leonardo","apellido":"Caballero"}
```

Luego de inicializar las variables del código anterior, usted puede usar la función *vars()* (página 122) para obtener un diccionario conteniendo ámbito actual de las variables, ejecutando:

```
>>> vars()
{'tupla': (11, 'Chao Plone', True, None),
'__builtins__': <module '__builtin__' (built-in)>,
'numero': 123456, '__package__': None, 'cadena': 'Hola Plone',
'diccionario': {'apellido': 'Caballero', 'nombre': 'Leonardo'},
'__name__': '__main__', 'lista': [7, 8, 9, 0], '__doc__': None}
```

Si desea eliminar la referencia a la variable `cadena`, ejecuta:

```
>>> del cadena
>>> vars()
{'tupla': (11, 'Chao Plone', True, None),
'__builtins__': <module '__builtin__' (built-in)>,
'numero': 123456, '__package__': None,
'diccionario': {'apellido': 'Caballero', 'nombre': 'Leonardo'},
'__name__': '__main__', 'lista': [7, 8, 9, 0], '__doc__': None}
```

Como pudo ver en el ejemplo anterior que elimino la referencia a la variable `cadena`, incluso al volver a la función *vars()* (página 122) ya no sale en el ámbito de variables disponibles.

La eliminación de una lista de objetivos elimina recursivamente cada objetivo, de izquierda a derecha.

```
>>> del numero, lista, tupla, diccionario
>>> vars()
{'__builtins__': <module '__builtin__' (built-in)>,
'__package__': None, '__name__': '__main__', '__doc__': None}
```

Como pudo ver en el ejemplo anterior que elimino las referencias a las variables `numero`, `lista`, `tupla`, `diccionario` que incluso al volver a la función *vars()* (página 122) ya no están en el ámbito de variables disponibles.

La eliminación de un nombre elimina el enlace de ese nombre del espacio de nombres *local* o *global*, dependiendo de si el nombre aparece en una sentencia «*global* (página 31)» en el mismo bloque de código. Si el nombre no está vinculado, se generará una excepción «*NameError* (página 192)».

Truco: Es ilegal eliminar un nombre del espacio de nombres *local* si aparece como una variable libre en un bloque anidado.

La eliminación de las referencias de atributos, suscripciones y segmentaciones se pasa al objeto primario involucrado; la eliminación de un corte es en general equivalente a la asignación de un corte vacío del tipo correcto (pero incluso esto está determinado por el objeto cortado).

3.2.6 Sentencia global

La sentencia `global` es una declaración que se mantiene para todo el bloque de código actual. Eso significa que los identificadores listados son interpretados como globales. Eso podría ser imposible asignar a una variable global sin la sentencia `global`, aunque las variables libres pueden referirse a globales sin ser declaradas globales.

```
>>> variable1 = "variable original"
>>> def variable_global():
...     global variable1
...     variable1 = "variable global modificada"
...
>>> print variable1
variable original
>>> variable_global()
>>> print variable1
variable global modificada
```

Como se puede ver, después de llamar a la función `variable_global()`, la variable `variable1` queda modificada. En general, este procedimiento debe utilizarse con precaución.

Importante: Usted puede descargar el código usado en esta sección haciendo clic en los siguientes enlaces: `constantes.py` y `main.py`.

Truco: Para ejecutar el código `constantes.py` y `main.py`, abra una consola de comando, acceda al directorio donde se encuentra ambos programas:

```
leccion3/
├── constantes.py
└── main.py
```

Si tiene la estructura de archivo previa, entonces ejecute el siguiente comando:

```
python main.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.3 Operadores de asignaciones

Los operadores de asignación se utilizan para

Existe en Python todo un grupo de operadores los cuales le permiten básicamente asignar un valor a una variable, usando el operador «=». Con estos operadores pueden aplicar la técnica denominada *asignación aumentada* (página 34).

3.3.1 Operador =

El operador *igual a*, (=), es el más simple de todos y asigna a la variable del lado izquierdo cualquier variable o resultado del lado derecho.

3.3.2 Operador +=

El operador += suma a la variable del lado izquierdo el valor del lado derecho.

```
>>> r = 5; r += 10; r
15
```

En el ejemplo anterior si la variable «r» es igual a 5 y r += 10, entonces la variable «r» sera igual a 15. Su equivalente seria el siguiente:

```
>>> r = 5; r = r + 10; r
15
```

3.3.3 Operador -=

El operador −= resta a la variable del lado izquierdo el valor del lado derecho.

```
>>> r = 5; r -= 10; r
-5
```

En el ejemplo anterior si la variable «r» es igual a 5 y r -= 10, entonces la variable «r» sera igual a −5. Su equivalente seria el siguiente:

```
>>> r = 5; r = r - 10; r
-5
```

3.3.4 Operador *=

El operador ⋆= multiplica a la variable del lado izquierdo el valor del lado derecho.

```
>>> r = 5; r *= 10; r
50
```

En el ejemplo anterior si la variable «r» es igual a 5 y r *= 10, entonces la variable «r» sera igual a 50. Su equivalente seria el siguiente:

```
>>> r = 5; r = r * 10; r
50
```

3.3.5 Operador /=

El operador /= divide a la variable del lado izquierdo el valor del lado derecho.

```
>>> r = 5; r /= 10; r
0
```

En el ejemplo anterior si la variable «r» es igual a 5 y r /= 10, entonces la variable «r» sera igual a 0. Su equivalente seria el siguiente:

```
>>> r = 5; r = r / 10; r
0
```

3.3.6 Operador **=

El operador ⋆⋆= calcula el exponente a la variable del lado izquierdo el valor del lado derecho.

```
>>> r = 5; r **= 10; r
9765625
```

En el ejemplo anterior si la variable «r» es igual a 5 y r **= 10, entonces la variable «r» sera igual a 9765625. Su equivalente seria el siguiente:

```
>>> r = 5; r = r ** 10; r
9765625
```

3.3.7 Operador //=

El operador //= calcula la división entera a la variable del lado izquierdo el valor del lado derecho.

```
>>> r = 5; r //= 10; r
0
```

En el ejemplo anterior si la variable «r» es igual a 5 y r //= 10, entonces la variable «r» sera igual a 0. Su equivalente seria el siguiente:

```
>>> r = 5; r = r // 10; r
0
```

3.3.8 Operador %=

El operador %= devuelve el resto de la división a la variable del lado izquierdo el valor del lado derecho.

```
>>> r = 5; r %= 10; r
5
```

En el ejemplo anterior si la variable «r» es igual a 5 y r %= 10, entonces la variable «r» sera igual a 5. Su equivalente seria el siguiente:

```
>>> r = 5; r = r % 10; r
5
```

3.3.9 Asignación aumentada

Es frecuente que una variable tenga que ser definida de nuevo en función de sí misma. Normalmente usted escribir la siguiente sintaxis:

```
>>> contador = contador + 1
```

El código anterior, se puede abreviar a su equivalente, usando la asignación aumentada, de la siguiente manera:

```
>>> contador += 1
```

El código anterior, no sólo es más corto de escribir, sino también más eficiente en tiempo de ejecución.

3.3.10 Ejemplos

A continuación, se presentan algunos ejemplos de su uso:

```
a, b, c = 21, 10, 0

print ("Valor de variable 'a':", a)
print ("Valor de variable 'b':", b)

c = a + b
```

(continué en la próxima página)

(proviene de la página anterior)

```
print ("Operador = | El valor de variable 'c' es ", c)

c += a
print ("Operador += | El valor de variable 'c' es ", c)

c *= a
print ("Operador *= | El valor de variable 'c' es ", c)

c /= a
print ("Operador /= | El valor de variable 'c' es ", c)

c = 2
c %= a
print ("Operador %= | El valor de variable 'c' es ", c)

c **= a
print ("Operador **= | El valor de variable 'c' es ", c)

c //= a
print ("Operador //= | El valor de variable 'c' es ", c)
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `operadores_asignaciones.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python operadores_asignaciones.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.4 Operadores aritméticos

Los valores numéricos son además el resultado de una serie de operadores aritméticos y matemáticos:

3.4.1 Operador Suma

El operador + suma los valores de tipo de datos numéricos.

```
>>> 3 + 2
5
```

3.4.2 Operador Resta

El operador − resta los valores de tipo de datos numéricos.

```
>>> 4 - 7
-3
```

3.4.3 Operador Negación

El operador − asigna un valor negativo a un tipo de datos numéricos.

```
>>> -7
-7
```

3.4.4 Operador Multiplicación

El operador * multiplica los valores de tipo de datos numéricos.

```
>>> 2 * 6
12
```

3.4.5 Operador Exponente

El operador ** calcula el exponente entre valores de tipo de datos numéricos.

```
>>> 2 ** 6
64
```

3.4.6 Operador división

El operador *división* el resultado que se devuelve es un número real.

```
>>> 3.5 / 2
1.75
```

3.4.7 Operador división entera

El operador *división entera* el resultado que se devuelve es solo la parte entera.

```
>>> 3.5 // 22
1.0
```

No obstante hay que tener en cuenta que si utilizamos dos operandos enteros, Python determinará que quiere que la variable resultado también sea un entero, por lo que el resultado de, por ejemplo, `3 / 2` y `3 // 2` sería el mismo: `1`.

Si quisiéramos obtener los decimales necesitaríamos que al menos uno de los operandos fuera un número real, bien indicando los decimales:

```
r = 3.0 / 2
```

O bien utilizando la función *float()* (página 125) para convertir a entero coma flotante o real:

```
r = float(3) / 2
```

Esto es así porque cuando se mezclan tipos de números, Python convierte todos los operandos al tipo más complejo de entre los tipos de los operandos.

3.4.8 Operador Módulo

El operador *módulo* no hace otra cosa que devolver el resto de la división entre los dos operandos. En el ejemplo, `7 / 2` sería `3`, con `1` de resto, luego el *módulo* es `1`.

```
>>> 7 % 2
1
```

3.4.9 Orden de precedencia

El orden de precedencia de ejecución de los operadores aritméticos es:

1. Exponente: `**`
2. Negación: `-`
3. Multiplicación, División, División entera, Módulo: `*`, `/`, `//`, `%`
4. Suma, Resta: `+`, `-`

Eso quiere decir que se debe usar así:

```
>>> 2**1/12
0.1666666666666666
>>>
```

Más igualmente usted puede omitir este orden de precedencia de ejecución de los operadores aritméticos usando paréntesis `()` anidados entre cada nivel calculo, por ejemplo:

```
>>> 2**(1/12)
1.0594630943592953
>>>
```

3.4.10 Ejemplos

A continuación, se presentan algunos ejemplos de su uso:

Ejemplo de definir variables numéricas

```
a, b, c, d = 26, 11.3, 5, 3.5
```

Ejemplo de operador aritmético Suma, Añade valores a cada lado del operador.

```
print (a + b)
```

Ejemplo de operador aritmético Resta, Resta el operando de la derecha del operador del lado izquierdo.

```
print (c - a)
```

Ejemplo de operador aritmético Multiplicación, Multiplica los valores de ambos lados del operador.

```
print (d * c)
```

Ejemplo de operador aritmético Exponente, Realiza el cálculo exponencial (potencia) de los operadores.

```
print (c ** 2)
```

Ejemplo de operador aritmético División.

```
print (float(c) / a)
```

Ejemplo de operador aritmético División entera.

```
print (7 / 3)
```

Ejemplo de operador aritmético Cociente de una división, la división de operandos que el resultado es el cociente en el cual se eliminan los dígitos después del punto decimal.

```
print (a // c)
```

Ejemplo de operador aritmético Módulo, el cual divide el operando de la izquierda por el operador del lado derecho y devuelve el resto.

```
print (7 % 3)
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `operadores_aritmeticos.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python operadores_aritmeticos.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.5 Operadores relacionales

Los valores booleanos son además el resultado de expresiones que utilizan operadores relacionales (comparaciones entre valores):

3.5.1 Operador ==

El operador == evalúa que los valores sean *iguales* para varios tipos de datos.

```
>>> 5 == 3
False
>>> 5 == 5
True
>>> "Plone" == 5
False
>>> "Plone" == "Plone"
True
>>> type("Plone") == str
True
```

3.5.2 Operador !=

El operador ! = evalúa si los valores son *distintos*.

```
>>> 5 != 3
True
>>> "Plone" != 5
True
>>> "Plone" != False
True
```

3.5.3 Operador <

El operador < evalúa si el valor del lado izquierdo es *menor que* el valor del lado derecho.

```
>>> 5 < 3
False
```

3.5.4 Operador >

El operador > evalúa si el valor del lado izquierdo es *mayor que* el valor del lado derecho.

```
>>> 5 > 3
True
```

3.5.5 Operador <=

El operador <= evalúa si el valor del lado izquierdo es *menor o igual que* el valor del lado derecho.

```
>>> 5 <= 3
False
```

3.5.6 Operador >=

El operador >= evalúa si el valor del lado izquierdo es *mayor o igual que* el valor del lado derecho.

```
>>> 5 >= 3
True
```

3.5.7 Ejemplos

A continuación, se presentan algunos ejemplos de su uso:

Ejemplo de definir variables de tipo numéricas, cadenas y listas

```
a, b, a1, b1, c1 = 5, 5, 7, 3, 3
cadena1, cadena2 = 'Hola', 'Adiós'
lista1, lista2 = [1, 'Lista Python', 23], [11, 'Lista Python', 23]
```

Ejemplo de operador relacional Igual

```
c = a == b
print (c)
```

Ejemplo de operador relacional Diferente

```
d = a1 != b
print (d)
```

Ejemplo de operador relacional Menor que

```
f = b1 < a1
print (f)
```

Ejemplo de operador relacional Mayor que

```
e = a1 > b1
print (e)
```

Ejemplo de operador relacional Menor o igual que

```
h = b1 <= c1
print (h)
```

Ejemplo de operador relacional Mayor o igual que

```
g = b1 >= c1
print (g)
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `operadores_relacionales.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python operadores_relacionales.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.6 Tipo números

Estos tipos de datos se crean mediante literales numéricos y se devuelven como resultados por operadores aritméticos y funciones aritméticas integradas. Los objetos numéricos son inmutables; Una vez creado su valor nunca cambia.

Por supuesto, los números de Python están fuertemente relacionados con los números matemáticos, pero están sujetos a las limitaciones de la representación numérica en las computadoras.

Python distingue entre enteros, números de punto flotante y números complejos:

Clase	Tipo	Notas	Ejemplo
`int`	Números	Número entero con precisión fija.	`42`
`long`	Números	Número entero en caso de overflow.	`42L` ó `45696678615198764 3L`
`float`	Números	Coma flotante de doble precisión.	`3.1415927`
`complex`	Números	Parte real y parte imaginaria *j*.	`(4.5 + 3j)`

3.6.1 Enteros

Los números enteros son aquellos que no tienen decimales, tanto positivos como negativos (además del cero). En Python se pueden representar mediante el tipo `int` (de integer, entero) o el tipo `long` (largo). La única diferencia es que el tipo long permite almacenar números más grandes. Es aconsejable no utilizar el tipo `long` a menos que sea necesario, para no malgastar memoria.

El tipo `int` de Python se implementa a bajo nivel mediante un tipo `long` de C. Y dado que Python utiliza C por debajo, como C, y a diferencia de Java, el rango de los valores que puede representar depende de la plataforma. En la mayor parte de las máquinas el `long` de C se almacena utilizando 32 bits, es decir, mediante el uso de una variable de tipo `int` de Python puede almacenar números de -2^{31} a $2^{31} - 1$, o lo que es lo mismo, de -2.147.483.648 a 2.147.483.647. En plataformas de 64 bits, el rango es de -9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807.

Ejemplo de enteros

A continuación, se presentan un ejemplo de su uso:

Ejemplo de definición de un tipo entero

```
entero = 7
print (entero, type(entero))
```

3.6.2 Enteros long

El tipo `long` de Python permite almacenar números de cualquier precisión, limitado por la memoria disponible en la máquina.

Al asignar un número a una variable esta pasará a tener tipo `int`, a menos que el número sea tan grande como para requerir el uso del tipo `long`.

```
>>> entero = 23
>>> type(entero)
<type 'int'>
```

También puede indicar a Python que un número se almacene usando `long` añadiendo una *L* al final:

```
>>> entero = 23L
>>> type(entero)
<type 'long'>
```

El literal que se asigna a la variable también se puede expresar como un `octal`, anteponiendo un cero:

```
# 027 octal = 23 en base 10
entero = 027
```

o bien en hexadecimal, anteponiendo un `0x`:

```
# 0x17 hexadecimal = 23 en base 10
entero = 0x17
```

Ejemplo de enteros long

A continuación, se presentan un ejemplo de su uso:

Ejemplo de definición de un tipo entero long

```
enterol = 23L
print (enterol, type(enterol))
```

3.6.3 Coma flotante

Los números reales son los que tienen decimales. En Python se expresan mediante el tipo `float`. En otros lenguajes de programación, como C, tiene también el tipo `double`, similar a `float` pero de mayor precisión (double = doble precisión).

Python, sin embargo, implementa su tipo `float` a bajo nivel mediante una variable de tipo `double` de C, es decir, utilizando 64 bits, luego en Python siempre se utiliza doble precisión, y en concreto se sigue el estándar IEEE 754: 1 bit para el signo, 11 para el exponente, y 52 para la mantisa. Esto significa que los valores que puede representar van desde $\pm$2,2250738585072020 x 10^{-308} hasta $\pm$1,7976931348623157×10^{308}.

La mayor parte de los lenguajes de programación siguen el mismo esquema para la representación interna. Pero como muchos sabréis esta tiene sus limitaciones, impuestas por el hardware.

Por eso desde Python 2.4 cuenta también con un nuevo tipo Decimal[57], para el caso de que se necesite representar fracciones de forma más precisa. Sin embargo este tipo está fuera del alcance de este tutorial, y sólo es necesario para el ámbito de la programación científica y otros relacionados.

Para aplicaciones normales puedes utilizar el tipo `float` sin miedo, como ha venido haciéndose desde hace años, aunque teniendo en cuenta que los números en coma flotante no son precisos (ni en este ni en otros lenguajes de programación).

Para representar un número real en Python se escribe primero la parte entera, seguido de un punto y por último la parte decimal.

```
real = 0.2703
```

También se puede utilizar notación científica, y añadir una e (de exponente) para indicar un exponente en base 10. Por ejemplo:

```
real = 0.1e-3
```

sería equivalente a 0.1 x 10^{-3} = 0.1 x 0.001 = 0.0001

Ejemplo de enteros float

A continuación, se presentan un ejemplo de su uso:

Ejemplo de definir tipo entero coma flotante

```
float_1, float_2, float_3 = 0.348, 10.5, 1.5e2
print (float_1, type(float_1))
print (float_2, type(float_2))
print (float_3, type(float_3))
```

Ejemplo de definir tipo entero coma flotante con exponente en base 10

```
real = 0.56e-3
print (real, type(real))
```

[57] https://www.python.org/dev/peps/pep-0327/

3.6.4 Complejos

Los números complejos son aquellos que tienen parte imaginaria. Si no conocías de su existencia, es más que probable que nunca lo vayas a necesitar, por lo que puede saltarte este apartado tranquilamente.

De hecho la mayor parte de lenguajes de programación carecen de este tipo, aunque sea muy utilizado por ingenieros y científicos en general.

En el caso de que necesite utilizar números complejos, o simplemente tiene curiosidad, este tipo, llamado `complex` en Python, también se almacena usando coma flotante, debido a que estos números son una extensión de los números reales.

En concreto se almacena en una estructura de C, compuesta por dos variables de tipo `double`, sirviendo una de ellas para almacenar la parte real y la otra para la parte imaginaria.

Los números complejos en Python se representan de la siguiente forma:

```
complejo = 2.1 + 7.8j
```

Ejemplo de enteros complex

A continuación, se presentan un ejemplo de su uso:

Ejemplo de definición de tipo entero complejos

```
complejo = 3.14j
print (complejo, complejo.imag, complejo.real, type(complejo))
```

3.6.5 Convertir a numéricos

Para convertir a *tipos numéricos* (página 40) debe usar las siguientes *funciones integradas* (página 113) en el interprete Python:

- La función *int()* (página 126) devuelve un tipo de datos *número entero* (página 41).
- La función *long()* (página 126) devuelve un tipo de datos *número entero long* (página 41).
- La función *float()* (página 125) devuelve un tipo de datos *número entero float* (página 42).
- La función *complex()* (página 125) devuelve un tipo de datos *número complejo* (página 43).

3.6.6 Ayuda integrada

Usted puede consultar toda la documentación disponible sobre las **números enteros** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(int)
```

Para salir de esa ayuda presione la tecla `q`.

Usted puede consultar toda la documentación disponible sobre las **números enteros long** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(long)
```

Para salir de esa ayuda presione la tecla `q`.

Usted puede consultar toda la documentación disponible sobre las **números coma flotante** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(float)
```

Para salir de esa ayuda presione la tecla `q`.

Usted puede consultar toda la documentación disponible sobre las **números complejos** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(complex)
```

Para salir de esa ayuda presione la tecla `q`.

Truco: Para más información consulte las funciones integradas para *operaciones numéricas* (página 124).

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `tipo_numericos.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python tipo_numericos.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.7 Tipo booleanos

El tipo booleano sólo puede tener dos valores: `True` (verdadero) y `False` (falso). Estos valores son especialmente importantes para las expresiones condicionales y los bucles, como verá más adelante.

Clase	Tipo	Notas	Ejemplo
`bool`	Números	Valor booleano falso.	`False`
`bool`	Números	Valor booleano verdadero.	`True`

En el contexto de las operaciones booleanas, y también cuando las expresiones son usadas bajo sentencias de flujo de control, los siguientes valores son interpretados como `False`:

- `False`.
- *None* (página 210).
- *Número cero* (página 40) en todos los tipos.
- *Cadena de caracteres* (página 46) vaciás.
- Contenedores, incluyendo *cadenas de caracteres*, *tuplas* (página 63), *listas* (página 58), *diccionarios* (página 65) y *conjuntos* (página 75) mutables e inmutables.

A continuación, varios ejemplos en códigos de los citado previamente:

```
>>> False
False
>>> False == False
True
>>> 0 == False
True
>>> "" == False
False
>>> None == False
False
>>> [] == False
False
>>> () == False
False
>>> {} == False
False
>>> ['', ''] == False
False
```

Todos los otros valores son interpretados por defecto a `True`. El operador lógico *not* (página 86) produce `True` si su argumento es falso, `False` de lo contrario.

Los tipos integrados `False` y `True` son solamente dos instancias de la clase `bool`. En realidad el tipo `bool` es una *subclase* (página 202) del tipo `int` o entero plano, es decir, sus valores son `0` y `1` respectivamente, en casi todos los contextos:

```
>>> int(False)
0
>>> int(True)
1
```

En el ejemplo anterior se convierte tipos booleanos a tipo enteros, siempre devuelve sus valores numéricos `0` y `1`. La excepción a la regla anterior sucede cuando un tipo booleano es convertido a un tipo de *cadenas de caracteres* (página 46), las cadenas *"False"* y/o *"True"* son retornadas, respectivamente:

```
>>> type(True)
<type 'bool'>
>>> str(True)
'True'
>>> type(str(True))
<type 'str'>
>>>
>>> type(False)
<type 'bool'>
>>> str(False)
'False'
>>> type(str(False))
<type 'str'>
```

Puede que esto para usted, no lo entienda mucho, si no conoces los términos de la *orientación a objetos* (página 194), que se tocará más adelante, aunque tampoco es nada importante.

Importante: Los tipos *booleanos* no puede ser a su vez una subclase.

3.7.1 Convertir a booleanos

Para convertir a *tipos booleanos* debe usar la función *bool()* (página 129) la cual *esta integrada* (página 113) en el interprete Python.

3.7.2 Ejemplos

A continuación, se presentan algunos ejemplos de su uso:

Ejemplo de tipos de datos booleanos

```
aT, aF = True, False
print ("El valor es", aT, "de tipo:", type(aT), "\n")
print ("El valor es", aF, "de tipo:", type(aF))
```

Ejemplo de operadores booleanos

```
aAnd = True and False
print ("SI es Verdadero Y Falso, es", aAnd, "de", type(aAnd), "\n")
aOr = True or False
print ("SI es Verdadero O Falso, es", aOr, "de", type(aOr), "\n")
aNot = not True
print ("Si NO es Verdadero, es", aNot, "de", type(aNot), "\n")
```

3.7.3 Ayuda integrada

Usted puede consultar toda la documentación disponible sobre los **booleanos** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(bool)
```

Para salir de esa ayuda presione la tecla `q`.

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `tipo_booleanos.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python tipo_booleanos.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.8 Tipo cadenas de caracteres

Las cadenas de caracteres, son secuencias inmutables que contienen caracteres encerrado entre comillas.

3.8.1 Cadenas cortas

Son caracteres encerrado entre comillas simples (') o dobles (").

```
>>> 'Hola Mundo'
'Hola Mundo'
```

3.8.2 Cadenas largas

Son caracteres encerrados entre grupo comillas triples simples (`'''`) o dobles (`"""`), están son generalmente son referenciadas como *cadenas de triple comillas*.

```
>>> """Clase que representa una Persona"""
'Clase que representa una Persona'
>>> '''Clase que representa un Supervisor'''
'Clase que representa un Supervisor'
```

3.8.3 Clases

A continuación, una lista de clases integradas Python para los tipos de cadenas de caracteres:

basestring

Es la *clase base* de las clases `str` y `unicode`.

str

Son *secuencias inmutables* de cadenas de caracteres con soporte a caracteres `ASCII`.

```
>>> 'Hola Mundo'
'Hola Mundo'
>>> "Hola Mundo"
'Hola Mundo'
```

unicode

Son *secuencias inmutables* de cadenas de caracteres con soporte a caracteres `Unicode`.

```
>>> u'Jekechitü'
u'Jekechit\xfc'
```

3.8.4 Prefijo de cadenas

Una cadena puede estar precedida por el carácter:

- `r`/`R`, el cual indica, que se trata de una cadena `raw` (del inglés, cruda). Las cadenas `raw` se distinguen de las normales en que los caracteres escapados mediante la barra invertida (`\`) no se sustituyen por sus contrapartidas. Esto es especialmente útil, por ejemplo, para usar las expresiones regulares.

  ```
  >>> raw = r"\t\nHola Plone\n"
  >>> type(raw)
  <type 'str'>
  ```

- `u`/`U`, el cual indica, que se trata de una cadena que utiliza codificación *unicode* (página 47).

```
>>> saber_mas = u"Atüjaa oo'omüin..."
>>> type(saber_mas)
<type 'unicode'>
>>> vocales = U"äóè"
>>> type(vocales)
<type 'unicode'>
```

3.8.5 Cadenas de escape

Para escapar caracteres dentro de cadenas de caracteres se usa el carácter \ seguido de cualquier carácter ASCII.

Secuencia Escape	Significado
\newline	Ignorado
\\	Backslash (\)
\'	Comillas simple (')
\"	Comillas doble (")
\a	Bell ASCII (BEL)
\b	Backspace ASCII (BS)
\f	Formfeed ASCII (FF)
\n	Linefeed ASCII (LF)
\N{name}	Carácter llamado *name* en base de datos Unicode (Solo Unicode)
\r	Carriage Return ASCII (CR)
\t	Tabulación Horizontal ASCII (TAB)
\uxxxx	Carácter con valor hex 16-bit *xxxx* (Solamente Unicode). Ver *hex* (página 126).
\Uxxxxxxxx	Carácter con valor hex 32-bit *xxxxxxxx* (Solamente Unicode). Ver *hex* (página 126).
\v	Tabulación Vertical ASCII (VT)
\ooo	Carácter con valor octal *ooo*. Ver *octal* (página 129).
\xhh	Carácter con valor hex *hh*. Ver *hex* (página 126).

También es posible encerrar una cadena entre triples comillas (simples o dobles). De esta forma puede escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea que se introdujeron sin tener que recurrir a los caracteres escapados y las comillas como los anteriores.

3.8.6 Operaciones

Las cadenas también admiten operadores aritméticos como los siguientes:

- El operador *suma* (página 35) para realizar concatenación de cadenas de caracteres:

```
>>> a, b = "uno", "dos"
>>> a + b
'unodos'
```

- El operador *multiplicación* (página 36) para repetir la cadena de caracteres por N veces definidas en la multiplicación:

```
>>> c = "tres"
>>> c * 3
'trestrestres'
```

- El operador *modulo* (página 37) usado la técnica de interpolación variables dentro de una cadena de caracteres. Más información consulte la sección *formateo %* (página 53).

3.8.7 Comentarios

Son cadenas de caracteres las cuales constituyen una ayuda esencial tanto para quien está desarrollando el programa, como para otras personas que lean el código.

Los comentarios en el código tienen una vital importancia en el desarrollo de todo programa, algunas de las funciones más importantes que pueden cumplir los comentarios en un programa, son:

- Brindar información general sobre el programa.
- Explicar qué hace cada una de sus partes.
- Aclarar y/o fundamentar el funcionamiento de un bloque específico de código, que no sea evidente de su propia lectura.
- Indicar cosas pendientes para agregar o mejorar.

El signo para indicar el comienzo de un comentario en Python es el carácter numeral #, a partir del cual y hasta el fin de la línea, todo se considera un comentario y es ignorado por el intérprete Python.

```
>>> # comentarios en linea
...
>>>
```

El carácter # puede estar al comienzo de línea (en cuyo caso toda la línea será ignorada), o después de finalizar una instrucción válida de código.

```
>>> # Programa que calcula la sucesión
... # de números Fibonacci
...
>>> # se definen las variables
... a, b = 0, 1
>>> while b < 100: # mientras b sea menor a 100 itere
...     print b,
...     a, b = b, a + b # se calcula la sucesión Fibonacci
...
1 1 2 3 5 8 13 21 34 55 89
```

Comentarios multilínea

Python no dispone de un método para delimitar bloques de comentarios de varias líneas.

Al igual que los comentarios de un sola linea, son cadenas de caracteres, en este caso van entre triples comillas (simples o dobles), esto tiene el inconveniente que, aunque no genera código ejecutable, el bloque delimitado no es ignorado por el intérprete Python, que crea el correspondiente objeto de tipo *cadena de caracteres* (página 46).

```
>>> """comentarios en varias lineas"""
'comentarios en varias lineas'
>>> '''comentarios en varias lineas'''
'comentarios en varias lineas'
```

A continuación, una comparación entre comentarios multilínea y comentarios en solo una linea:

```
>>> # Calcula la sucesión
... # de números Fibonacci
...
>>> """Calcula la sucesión
... de números Fibonacci"""
'Calcula la sucesi\xc3\xb3n \nde n\xc3\xbameros Fibonacci'
```

Entonces existen al menos dos (02) alternativas para introducir comentarios multilínea son:

- Comentar cada una de las líneas con el carácter #: en general todos los editores de programación y entornos de desarrollo (IDEs) disponen de mecanismos que permiten comentar y descomentar fácilmente un conjunto de líneas.
- Utilizar triple comillas (simples o dobles) para generar una cadena multilínea: si bien este método es aceptado.

A continuación, un ejemplo de Comentarios multilínea y de solo una linea:

```
>>> u"""Calcula la sucesiónde números Fibonacci"""
u'Calcula la sucesi\xf3nde n\xfameros Fibonacci'
>>> # se definen las variables
... a, b = 0, 1
>>> while b < 100:
...     print b,
...     # se calcula la sucesión Fibonacci
...     a, b = b, a + b
...
1 1 2 3 5 8 13 21 34 55 89
```

Los comentarios multilínea usado con mucha frecuencia como en las varias sintaxis Python como *comentarios de documentación* (página 50) a continuación se listan las sintaxis más comunes:

- *Módulos* (página 159).
- *Funciones* (página 100).
- *Clases* (página 200).
- *Métodos* (página 198).

3.8.8 Docstrings

En Python todos los objetos cuentan con una variable especial llamada `__doc__`, gracias a la cual puede describir para qué sirven los objetos y cómo se usan. Estas variables reciben el nombre de `docstrings`, o cadenas de documentación[58].

Ten en cuenta, una buena documentación siempre dará respuesta a las dos preguntas:

- ¿Para qué sirve?
- ¿Cómo se utiliza?

Funciones

Python implementa un sistema muy sencillo para establecer el valor de las `docstrings` en las funciones, únicamente tiene que crear un comentario en la primera línea después de la declaración.

```
>>> def hola(arg):
...     """El docstring de la función"""
...     print "Hola", arg, "!"
...
>>> hola("Plone")
Hola Plone !
```

Puede puede consultar la documentación de la función `hola()` debe utilizar la función integrada *help()* (página 118) y pasarle el argumento del objeto de función `hola()`:

```
>>> help(hola)

Help on function hola in module __main__:
```

(continué en la próxima página)

[58] http://docs.python.org.ar/tutorial/2/controlflow.html#tut-docstrings

(proviene de la página anterior)

```
hola(arg)
    El docstring de la función

>>>
>>> print hola.__doc__
El docstring de la función
```

Clases y métodos

De la misma forma puede establecer la documentación de la clase después de la definición, y de los métodos, como si fueran funciones:

```
>>> class Clase:
...
...     """El docstring de la clase"""
...     def __init__(self):
...         """El docstring del método constructor de clase"""
...
...     def metodo(self):
...         """El docstring del método de clase"""
...
>>> o = Clase()
>>> help(o)

Help on instance of Clase in module __main__:

class Clase
 |  El docstring de la clase
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |      El docstring del método constructor de clase
 |
 |  metodo(self)
 |      El docstring del método de clase

>>> o.__doc__
'El docstring de la clase'
>>> o.__init__.__doc__
'El docstring del método constructor de clase'
>>> o.metodo.__doc__
'El docstring del método de clase'
```

Scripts y módulos

Cuando tiene un script o módulo, la primera línea del mismo hará referencia al `docstrings` del módulo, en él debe explicar el funcionamiento del mismo:

En el archivo `mi_modulo.py` debe contener el siguiente código:

```
"""El docstring del módulo"""

def despedir():
    """El docstring de la función despedir"""
    print "Adiós! desde función despedir() del módulo prueba"
```

(continué en la próxima página)

(proviene de la página anterior)

```
def saludar():
    """El docstring de la función saludar"""
    print "Hola! desde función saludar() del módulo prueba"
```

Entonces, usted debe importar el módulo anterior, para consultar la documentación del módulo `mi_modulo` debe utilizar la función integrada *help()* (página 118) y pasarle el argumento el nombre de módulo `mi_modulo`, de la siguiente manera:

```
>>> import mi_modulo
>>> help(mi_modulo)

Help on module mi_modulo:

NAME
    mi_modulo - El docstring del módulo
FUNCTIONS
    despedir()
        El docstring de la función despedir
    saludar()
        El docstring de la función saludar
```

También puede consultar la documentación de la función `despedir()` dentro del módulo `mi_modulo`, usando la función integrada *help()* (página 118) y pasarle el argumento el formato *nombre_modulo.nombre_funcion*, es decir, `mi_modulo.despedir`, de la siguiente manera:

```
>>> help(mi_modulo.despedir)

Help on function despedir in module mi_modulo:

despedir()
    El docstring de la función despedir
```

Opcionalmente , usted puede listar las variables y funciones del módulo con la función `dir()`, de la siguiente manera:

```
>>> dir(mi_modulo)
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'despedir',
 'saludar']
```

Como puede apreciar, muchas de estas variables son especiales, puede comprobar sus valores:

```
>>> print mi_modulo.__name__      # Nombre del módulo
'mi_modulo'
>>> print mi_modulo.__doc__       # Docstring del módulo
'El docstring del módulo'
>>> print mi_modulo.__package__   # Nombre del paquete del módulo
```

3.8.9 Formateo de cadenas

Python soporta múltiples formas de formatear una cadena de caracteres. A continuación se describen:

Formateo %

El carácter modulo % es un operador integrado en Python. Ese es conocido como el operador de interpolación. Usted necesitará proveer el % seguido por el tipo que necesita ser formateado o convertido. El operador % entonces substituye la frase " %tipodato" con cero o mas elementos del tipo de datos especificado:

```
>>> tipo_calculo = "raíz cuadrada de dos"
>>> valor = 2**0.5
>>> print "el resultado de %s es %f" % (tipo_calculo, valor)
el resultado de raíz cuadrada de dos es 1.414214
```

También aquí se puede controlar el formato de salida. Por ejemplo, para obtener el valor con 8 dígitos después de la coma:

```
>>> tipo_calculo = "raíz cuadrada de dos"
>>> valor = 2**0.5
>>> print "el resultado de %s es %.8f" % (tipo_calculo, valor)
el resultado de raíz cuadrada de dos es 1.41421356
```

Con esta sintaxis hay que determinar el tipo del objeto:

- `%c` = str, simple carácter.
- `%s` = str, cadena de carácter.
- `%d` = int, enteros.
- `%f` = float, coma flotante.
- `%o` = octal.
- `%x` = hexadecimal.

A continuación un ejemplo por cada tipo de datos:

```
>>> print "CMS: %s, ¿Activar S o N?: %c" % ("Plone", "S")
CMS: Plone, ¿Activar S o N?: S
>>> print "N. factura: %d, Total a pagar: %f" % (345, 658.23)
N. factura: 345, Total a pagar: 658.230000
>>> print "Tipo Octal: %o, Tipo Hexadecimal: %x" % (027, 0x17)
Tipo Octal: 27, Tipo Hexadecimal: 17
```

Clase formatter

`formatter` es una de las clases integradas `string`. Ese provee la habilidad de hacer variable compleja de substituciones y formateo de valores usando el método *format()* (página 53). Es le permite crear y personalizar sus propios comportamientos de formatos de cadena de caracteres para reescribir los métodos públicos y contiene: `format()`, `vformat()`. Ese tiene algunos métodos que son destinado para ser remplazados por las sub-clases: `parse()`, `get_field()`, `get_value()`, `check_unused_args()`, `format_field()` y `convert_field()`.

format()

Este método devuelve una versión formateada de una cadena de caracteres, usando substituciones desde argumentos `args` y `kwargs`. Las substituciones son identificadas entre llaves `{ }` dentro de la cadena de caracteres (llamados campos de formato), y son sustituidos en el orden con que aparecen como argumentos de `format()`, contando a partir de cero (*argumentos posicionales*).

Esto es una forma más clara y elegante es referenciar objetos dentro de la misma cadena, y usar este *método* para sustituirlos con los objetos que se le pasan como argumentos.

```
>>> tipo_calculo = "raíz cuadrada de dos"
>>> valor = 2**0.5
>>> print "el resultado de {} es {}".format(tipo_calculo, valor)
el resultado de raíz cuadrada de dos es 1.41421356237
```

También se puede referenciar a partir de la posición de los valores utilizando índices:

```
>>> tipo_calculo = "raíz cuadrada de dos"
>>> valor = 2**0.5
>>> print "el resultado de {0} es {1}".format(tipo_calculo, valor)
el resultado de raíz cuadrada de dos es 1.41421356237
```

Los objetos también pueden ser referenciados utilizando un identificador con una clave y luego pasarla como argumento al método:

```
>>> tipo_calculo = "raíz cuadrada de dos"
>>> print "el resultado de {nombre} es {resultado}".format(
...     nombre=tipo_calculo, resultado=2**0.5)
el resultado de raíz cuadrada de dos es 1.41421356237
```

Formateo avanzado

Este método soporta muchas técnicas de formateo, aquí algunos ejemplos:

Alinear una cadena de caracteres a la derecha en 30 caracteres, con la siguiente sentencia:

```
>>> print "{:>30}".format("raíz cuadrada de dos")
     raíz cuadrada de dos
```

Alinear una cadena de caracteres a la izquierda en 30 caracteres (crea espacios a la derecha), con la siguiente sentencia:

```
>>> print "{:30}".format("raíz cuadrada de dos")
raíz cuadrada de dos
```

Alinear una cadena de caracteres al centro en 30 caracteres, con la siguiente sentencia:

```
>>> print "{:^30}".format("raíz cuadrada de dos")
    raíz cuadrada de dos
```

Truncamiento a 9 caracteres, con la siguiente sentencia:

```
>>> print "{:.9}".format("raíz cuadrada de dos")
raíz cua
```

Alinear una cadena de caracteres a la derecha en 30 caracteres con truncamiento de 9, con la siguiente sentencia:

```
>>> print "{:>30.9}".format("raíz cuadrada de dos")
                      raíz cua
```

Formateo por tipo

Opcionalmente se puede poner el signo de dos puntos después del número o nombre, y explicitar el tipo del objeto:

- `s` para cadenas de caracteres (tipo *str* (página 46)).
- `d` para números enteros (tipo *int* (página 40)).
- `f` para números de coma flotante (tipo *float* (página 42)).

Esto permite controlar el formato de impresión del objeto. Por ejemplo, usted puede utilizar la expresión `.4f` para determinar que un número de coma flotante (`f`) se imprima con cuatro dígitos después de la coma (`.4`).

```
>>> tipo_calculo = "raíz cuadrada de dos"
>>> valor = 2**0.5
>>> print "el resultado de {0} es {resultado:.4f}".format(
...     tipo_calculo, resultado=valor)
el resultado de raíz cuadrada de dos es 1.4142
```

Formateo de números enteros, rellenados con espacios, con las siguientes sentencias:

```
>>> print "{:4d}".format(10)
  10
>>> print "{:4d}".format(100)
 100
>>> print "{:4d}".format(1000)
1000
```

Formateo de números enteros, rellenados con ceros, con las siguientes sentencias:

```
>>> print "{:04d}".format(10)
0010
>>> print "{:04d}".format(100)
0100
>>> print "{:04d}".format(1000)
1000
```

Formateo de números flotantes, rellenados con espacios, con las siguientes sentencias:

```
>>> print "{:7.3f}".format(3.1415926)
  3.142
>>> print "{:7.3f}".format(153.21)
153.210
```

Formateo de números flotantes, rellenados con ceros, con las siguientes sentencias:

```
>>> print "{:07.3f}".format(3.1415926)
003.142
>>> print "{:07.3f}".format(153.21)
153.210
```

3.8.10 Convertir a cadenas de caracteres

Para convertir a *tipos cadenas de caracteres* debe usar la función *str()* (página 136) la cual *esta integrada* (página 113) en el interprete Python.

Truco: Para más información consulte las funciones integradas para *operaciones en cadenas de caracteres* (página 129).

3.8.11 Ejemplos

A continuación, se presentan algunos ejemplos de su uso:

Ejemplo de definir cadenas de caracteres con comillas simples

```
# Definir comillas simples
cadena1 = 'Texto entre comillas simples,'
```

Ejemplo de definir cadenas de caracteres con comillas dobles

```
# Definir comillas dobles
cadena2 = "Texto entre comillas dobles,"
```

Ejemplo de definir cadenas de caracteres con código escapes

```
# Definir cadena con código escapes
cadena3 = 'Texto entre \n\tcomillas simples,'
```

Ejemplo de definir cadenas de caracteres con varias lineas

```
# Definir cadena varias lineas
cadena4 = """Texto linea 1
linea 2
linea 3
linea 4
.
.
.
.
.
linea N"""
```

Ejemplo operadores de repetición de cadenas de caracteres

```
cadena5 = "Cadena" * 3
print (cadena5 + ",", type(cadena5))
```

Ejemplo operadores de concatenación de cadenas de caracteres

```
nombre, apellido = "Leonardo", "Caballero"
nombre_completo = nombre + " " + apellido
print (nombre_completo + ",", type(nombre_completo))
```

Ejemplo de medir tamaño de la cadena con función «len()»

```
print ("El tamaño de la cadena es:", len(nombre_completo))
```

Ejemplo de acceder a rango de la cadena

```
print ("Acceso a rango de cadena: ", nombre_completo[3:13])
```

Ejemplo de consulta de ayuda a la función len

```
>>> help(len)

Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or collection.
```

Ejemplo de consulta de ayuda a la clase int

```
>>> help(int)

Help on class int in module __builtin__:

class int(object)
 |  int(x=0) -> int or long
 |  int(x, base=10) -> int or long
```

(continué en la próxima página)

(proviene de la página anterior)

```
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is floating point, the conversion truncates towards zero.
 |  If x is outside the integer range, the function returns a long instead.
```

Ejemplo de consulta de ayuda del módulo

```
>>> import datetime
>>> help(datetime)

Help on built-in module datetime:

NAME
    datetime - Fast implementation of the datetime type.

FILE
    (built-in)

CLASSES
    __builtin__.object
        date
            datetime
```

3.8.12 Ayuda integrada

Usted puede consultar toda la documentación disponible sobre las *cadenas de caracteres* (página 47) desde la *consola interactiva* (página 15) de la siguiente manera:

```
>>> help(str)
```

Para salir de esa ayuda presione la tecla `q`.

Usted puede consultar toda la documentación disponible sobre las cadenas de caracteres *unicode* (página 47) desde la *consola interactiva* (página 15) de la siguiente manera:

```
>>> help(unicode)
```

Para salir de esa ayuda presione la tecla `q`.

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `tipo_cadenas.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python tipo_cadenas.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.9 Tipo listas

En Python tiene varios tipos de datos *compuestos* y dentro de las secuencias, están los tipos de *cadenas de caracteres* (página 46). Otro tipo muy importante de secuencia son las *listas*.

Entre las *secuencias*, el más versátil, es la *lista*, para definir una, usted debe escribir es entre corchetes, separando sus elementos con comas cada uno.

La lista en Python son variables que almacenan `arrays`, internamente cada posición puede ser un tipo de datos distinto.

```
>>> factura = ['pan', 'huevos', 100, 1234]
>>> factura
['pan', 'huevos', 100, 1234]
```

Las listas en Python son:

- **heterogéneas**: pueden estar conformadas por elementos de distintos tipo, incluidos otras listas.
- **mutables**: sus elementos pueden modificarse.

Una lista en Python es una estructura de datos formada por una secuencia ordenada de objetos.

Los elementos de una lista pueden accederse mediante su índice, siendo 0 el índice del primer elemento.

```
>>> factura[0]
'pan'
>>> factura[3]
1234
```

La función *len()* (página 118) devuelve la longitud de la lista (su cantidad de elementos).

```
>>> len(factura)
4
```

Los índices de una lista inicia entonces de **0** hasta el tamaño de la lista menos uno (`len(factura) - 1`):

```
>>> len(factura) - 1
3
```

Pueden usarse también índices negativos, siendo **-1** el índice del último elemento.

```
>>> factura[-1]
1234
```

Los índices negativos van entonces de **-1** (último elemento) a `-len(factura)` (primer elemento).

```
>>> factura[-len(factura)]
'pan'
```

A través de los índices, pueden cambiarse los elementos de una lista en el lugar.

```
>>> factura[1] = "carne"
>>> factura
['pan', 'carne', 100, 1234
```

De esta forma se cambia el valor inicial de un elemento de la lista lo cual hacen una la lista *mutable*

3.9.1 Métodos

El el objeto de tipo *lista* integra una serie de métodos integrados a continuación:

append()

Este método agrega un elemento al final de una lista.

```
>>> versiones_plone = [2.5, 3.6, 4, 5]
>>> print versiones_plone
[2.5, 3.6, 4, 5]
>>> versiones_plone.append(6)
>>> print versiones_plone
[2.5, 3.6, 4, 5, 6]
```

count()

Este método recibe un elemento como argumento, y cuenta la cantidad de veces que aparece en la lista.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> print "6 ->", versiones_plone.count(6)
6 -> 1
>>> print "5 ->", versiones_plone.count(5)
5 -> 1
>>> print "2.5 ->", versiones_plone.count(2.5)
2.5 -> 1
```

extend()

Este método extiende una lista agregando un iterable al final.

```
>>> versiones_plone = [2.1, 2.5, 3.6]
>>> print versiones_plone
[2.1, 2.5, 3.6]
>>> versiones_plone.extend([4])
>>> print versiones_plone
[2.1, 2.5, 3.6, 4]
>>> versiones_plone.extend(range(5,7))
>>> print versiones_plone
[2.1, 2.5, 3.6, 4, 5, 6]
```

index()

Este método recibe un elemento como argumento, y devuelve el índice de su primera aparición en la lista.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6, 4]
>>> print versiones_plone.index(4)
3
```

El método admite como argumento adicional un índice inicial a partir de donde comenzar la búsqueda, opcionalmente también el índice final.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6, 4]
>>> versiones_plone[2]
3.6
>>> print versiones_plone.index(4, 2)
3
>>> versiones_plone[3]
4
>>> print versiones_plone.index(4, 5)
6
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> versiones_plone[6]
4
```

El método devuelve un excepción *ValueError* (página 193) si el elemento no se encuentra en la lista, o en el entorno definido.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6, 4]
>>> print versiones_plone.index(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 9 is not in list
```

insert()

Este método inserta el elemento x en la lista, en el índice i.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> print versiones_plone
[2.1, 2.5, 3.6, 4, 5, 6]
>>> versiones_plone.insert(2, 3.7)
>>> print versiones_plone
[2.1, 2.5, 3.7, 3.6, 4, 5, 6]
```

pop()

Este método devuelve el último elemento de la lista, y lo borra de la misma.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> print versiones_plone.pop()
6
>>> print versiones_plone
[2.1, 2.5, 3.6, 4, 5]
```

Opcionalmente puede recibir un argumento numérico, que funciona como índice del elemento (por defecto, -1)

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> print versiones_plone.pop(2)
3.6
>>> print versiones_plone
[2.1, 2.5, 4, 5, 6]
```

remove()

Este método recibe como argumento un elemento, y borra su primera aparición en la lista.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> print versiones_plone
[2.1, 2.5, 3.6, 4, 5, 6]
>>> versiones_plone.remove(2.5)
>>> print versiones_plone
[2.1, 3.6, 4, 5, 6]
```

El método devuelve un excepción *ValueError* (página 193) si el elemento no se encuentra en la lista.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> print versiones_plone
[2.1, 2.5, 3.6, 4, 5, 6]
>>> versiones_plone.remove(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

reverse()

Este método invierte el orden de los elementos de una lista.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> print versiones_plone
[2.1, 2.5, 3.6, 4, 5, 6]
>>> versiones_plone.reverse()
>>> print versiones_plone
[6, 5, 4, 3.6, 2.5, 2.1]
```

sort()

Este método ordena los elementos de una lista.

```
>>> versiones_plone = [4, 2.5, 5, 3.6, 2.1, 6]
>>> print versiones_plone
[4, 2.5, 5, 3.6, 2.1, 6]
>>> versiones_plone.sort()
>>> print versiones_plone
[2.1, 2.5, 3.6, 4, 5, 6]
```

El método `sort()` admite la opción `reverse`, por defecto, con valor `False`. De tener valor `True`, el ordenamiento se hace en sentido inverso.

```
>>> versiones_plone.sort(reverse=True)
>>> print versiones_plone
[6, 5, 4, 3.6, 2.5, 2.1]
```

3.9.2 Convertir a listas

Para convertir a *tipos listas* debe usar la función *list()* (página 139) la cual *esta integrada* (página 113) en el interprete Python.

Truco: Para más información consulte las funciones integradas para *operaciones de secuencias* (página 137).

3.9.3 Ejemplos

A continuación, se presentan algunos ejemplos de su uso:

Ejemplo de definir una colección ordenada/arreglos o vectores

```
# Definir una colección ordenada / arreglos o vectores
lista = [2, "CMS", True, ["Plone", 10]]
```

Ejemplo de acceder a un elemento especifico de una lista

```
# Acceder a un elemento especifico
l2 = lista[1]
```

Ejemplo de acceder a un elemento en una lista anidada

```
# Acceder a un elemento en una lista anidada
l3 = lista[3][0]
```

Ejemplo de definir nuevo valor de un elemento de lista

```
# Definir nuevo valor de un elemento de lista
lista[1] = 4
```

Ejemplo de obtener un rango de elemento especifico

```
# Obtener un rango de elemento especifico
l3 = lista[0:3]
```

Ejemplos de obtener un rango con saltos de elementos específicos

```
# Obtener un rango con saltos de elementos específicos
l4 = lista[0:3:2]
```

Ejemplo de iterar sobre cualquier secuencia

Usted puede iterar sobre cualquier secuencia (cadenas de caracteres, lista, claves en un diccionario, lineas en un archivo, . . .):

Ejemplo de iterar sobre una cadenas de caracteres

```
>>> vocales = 'aeiou'
>>> for letra in 'hermosa':
...     if letra in vocales:
...         print letra,
e o a
```

Ejemplo de iterar sobre una lista

Para separar una cadena en frases, los valores pueden separarse con la función integrada `split()`.

```
>>> mensaje = "Hola, como estas tu?"
>>> mensaje.split() # retorna una lista
['Hola,', 'como', 'estas', 'tu?']
>>> for palabra in mensaje.split():
...     print palabra
...
Hola,
como
estas
tu?
```

Ejemplo de iterar sobre dos o más secuencias

Para iterar sobre dos o más secuencias al mismo tiempo, los valores pueden emparejarse con la función integrada *zip()* (página 140).

```
>>> preguntas = ['nombre', 'objetivo', 'sistema operativo']
>>> respuestas = ['Leonardo', 'aprender Python y Plone', 'Linux']
>>> for pregunta, respuesta in zip(preguntas, respuestas):
...     print '¿Cual es tu {0}?, la respuesta es: {1}.'.format(
...         pregunta, respuesta)
```

(continué en la próxima página)

(proviene de la página anterior)

```
...
¿Cual es tu nombre?, la respuesta es: Leonardo.
¿Cual es tu objetivo?, la respuesta es: aprender Python y Plone.
¿Cual es tu sistema operativo?, la respuesta es: Linux.
```

3.9.4 Ayuda integrada

Usted puede consultar toda la documentación disponible sobre las **listas** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(list)
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `tipo_listas.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python tipo_listas.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.10 Tipo tuplas

Las tuplas son objetos de tipo *secuencia*, específicamente es un tipo de dato *lista* (página 58) inmutable. Esta no puede modificarse de ningún modo después de su creación.

3.10.1 Métodos

Son muy similares a las *listas* (página 58) y comparten varias de sus funciones y métodos integrados, aunque su principal diferencia es que son inmutables. El objeto de tipo *tupla* integra una serie de métodos integrados a continuación:

count()

Este método recibe un elemento como argumento, y cuenta la cantidad de veces que aparece en la tupla.

```
>>> valores = ("Python", True, "Zope", 5)
>>> print "True ->", valores.count(True)
True -> 1
>>> print "'Zope' ->", valores.count('Zope')
'Zope' -> 1
>>> print "5 ->", valores.count(5)
5 -> 1
```

index()

Comparte el mismo método *index()* (página 59) del tipo lista. Este método recibe un elemento como argumento, y devuelve el índice de su primera aparición en la tupla.

```
>>> valores = ("Python", True, "Zope", 5)
>>> print valores.index(True)
1
>>> print valores.index(5)
3
```

El método devuelve un excepción *ValueError* (página 193) si el elemento no se encuentra en la tupla, o en el entorno definido.

```
>>> valores = ("Python", True, "Zope", 5)
>>> print valores.index(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
```

3.10.2 Convertir a tuplas

Para convertir a *tipos tuplas* debe usar la función *tuple()* (página 139), la cual *está integrada* (página 113) en el interprete Python.

Truco: Para más información consulte las funciones integradas para *operaciones de secuencias* (página 137).

3.10.3 Ejemplos

A continuación, se presentan algunos ejemplos de su uso:

Ejemplo de definir simple de tupla

```
tupla = 12345, 54321, 'hola!'
```

Ejemplo de definir tuplas anidadas

```
otra = tupla, (1, 2, 3, 4, 5)
```

Operación asignar de valores de una tupla en variables

```
x, y, z = tupla
```

Cuidar seguimiento del número de la numeración

Una tarea común es iterar sobre una secuencia mientras cuidas el seguimiento de la numeración de un elemento.

Podría usar un bucle `while` con un contador o un bucle `for` usando la función *range()* (página 120) y la función *len()* (página 118):

```
>>> tecnologias = ('Zope', 'Plone', 'Pyramid')
>>> for i in range(0, len(tecnologias)):
...     print i, tecnologias[i]
...
0 Zope
1 Plone
2 Pyramid
```

Pero, Python provee la palabra reservada `enumerate` para esto:

```
print ("\nIterar tupla con función enumerate")
print ("==================================\n")

for index, item in enumerate(conexion_completa):
    print (index, item)
```

Caso real de conexión a BD

A continuación, un ejemplo más apegado a la realidad que busca establecer una conexión a una BD:

```
print ("\nDefiniendo conexión a BD MySQL")
print ("==============================\n")

conexion_bd = "127.0.0.1","root","qwerty","nomina",
print ("Conexión típica:", conexion_bd)
print (type(conexion_bd))
conexion_completa = conexion_bd, "3307","10",
print ("\nConexión con parámetros adicionales:", conexion_completa)
print (type(conexion_completa))

print ("\n")

print ("IP de la BD:",  conexion_completa[0][0])
print ("Usuario de la BD:",  conexion_completa[0][1])
print ("Contraseña de la BD:",  conexion_completa[0][2])
print ("Nombre de la BD:",  conexion_completa[0][3])
print ("Puerto de conexión:", conexion_completa[1])
print ("Tiempo de espera en conexión:", conexion_completa[2])

print ("""\nMás información acerca de MySQL y Python \
http://mysql-python.sf.net/MySQLdb.html\n""")
```

3.10.4 Ayuda integrada

Usted puede consultar toda la documentación disponible sobre las **tuplas** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(tuple)
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `tipo_tuplas.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python tipo_tuplas.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.11 Tipo diccionarios

El diccionario, define una relación uno a uno entre claves y valores.

Clase	Tipo	Notas	Ejemplo
`dict`	Mapeos	Mutable, sin orden.	`{'cms':"Plone", 'version':5}`

Un objeto *mapping* mapea valores *hashable* a objetos arbitrariamente. Los objetos Mapeos son objetos mutable. El **diccionario** es el único tipo de mapeo estándar actual. Para otro contenedores ver los integrados en las clases «*lista* (página 58)», «*conjuntos* (página 75)», y «*tupla* (página 63)», y el modulo «`collections`».

Los diccionarios pueden ser creados colocando una lista separada por coma de pares «key:value» entre `{}`, por ejemplo: «`{'python': 27, 'plone': 51}`» o «`{27:'python', 51:'plone'}`», o por el constructor «*dict()* (página 138)».

```
>>> diccionario = {
...     "clave1":234,
...     "clave2":True,
...     "clave3":"Valor 1",
...     "clave4":[1,2,3,4]
... }
>>> print diccionario, type(diccionario)
{'clave4': [1, 2, 3, 4], 'clave1': 234,
'clave3': 'Valor 1', 'clave2': True} <type 'dict'>
```

Usted puede acceder a los valores del diccionario usando cada su clave, se presenta unos ejemplos a continuación:

```
>>> diccionario['clave1']
234
>>> diccionario['clave2']
True
>>> diccionario['clave3']
'Valor 1'
>>> diccionario['clave4']
[1, 2, 3, 4]
```

Un diccionario puede almacenar los diversos tipos de datos integrados en Python usando la función *type()* (página 122), usted puede pasar el diccionario con la clave que usted desea determinar el tipo de dato, se presenta unos ejemplos a continuación:

```
>>> type(diccionario['clave1'])
<type 'int'>
>>> type(diccionario['clave2'])
<type 'bool'>
>>> type(diccionario['clave3'])
<type 'str'>
>>> type(diccionario['clave4'])
<type 'list'>
```

3.11.1 Operaciones

Los objetos de tipo **diccionario** permite una serie de operaciones usando operadores integrados en el interprete Python para su tratamiento, a continuación algunos de estos:

Acceder a valor de clave

Esta operación le permite acceder a un valor especifico del *diccionario* mediante su clave.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1, django=2.1)
>>> versiones['zope']
2.13
```

Asignar valor a clave

Esta operación le permite asignar el valor especifico del *diccionario* mediante su clave.

```
>>> versiones = {'python': 2.7, 'zope': 2.13, 'plone': None}
>>> versiones['plone']
>>> versiones['plone'] = 5.1
>>> versiones
{'python': 2.7, 'zope': 2.13, 'plone': 5.1}
>>> versiones['plone']
5.1
```

Iteración in

Este operador es el mismo operador integrado *in* (página 84) en el interprete Python pero aplicada al uso de la secuencia de tipo **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1, django=2.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1, 'django': 2.1}
>>> 'plone' in versiones
True
>>> 'flask' in versiones
False
```

En el ejemplo anterior este operador devuelve `True` si la clave esta en el diccionario `versiones`, de lo contrario devuelve `False`.

3.11.2 Métodos

Los objetos de tipo **diccionario** integra una serie de métodos integrados a continuación:

clear()

Este método remueve todos los elementos desde el **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.clear()
>>> print versiones
{}
```

copy()

Este método devuelve una copia superficial del tipo **diccionario**:

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> otro_versiones = versiones.copy()
>>> versiones == otro_versiones
True
```

fromkeys()

Este método crea un nuevo **diccionario** con *claves* a partir de un tipo de dato *secuencia*. El valor de `value` por defecto es el tipo *None* (página 210).

```
>>> secuencia = ('python', 'zope', 'plone')
>>> versiones = dict.fromkeys(secuencia)
>>> print "Nuevo Diccionario : %s" %  str(versiones)
Nuevo Diccionario : {'python': None, 'zope': None, 'plone': None}
```

En el ejemplo anterior inicializa los valores de cada clave a `None`, mas puede inicializar un *valor* común por defecto para cada *clave*:

```
>>> versiones = dict.fromkeys(secuencia, 0.1)
>>> print "Nuevo Diccionario : %s" %  str(versiones)
Nuevo Diccionario : {'python': 0.1, 'zope': 0.1, 'plone': 0.1}
```

get()

Este método devuelve el valor en base a una coincidencia de búsqueda en un diccionario mediante una clave, de lo contrario devuelve el objeto *None* (página 210).

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.get('plone')
5.1
>>> versiones.get('php')
>>>
```

has_key()

Este método devuelve el valor `True` si el diccionario tiene presente la clave enviada como argumento.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.has_key('plone')
True
>>> versiones.has_key('django')
False
```

items()

Este método devuelve una lista de pares de diccionarios (clave, valor), como 2 tuplas.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.items()
[('zope', 2.13), ('python', 2.7), ('plone', 5.1)]
```

iteritems()

Este método devuelve un *iterador* (página 92) sobre los elementos (clave, valor) del diccionario. Lanza una excepción *StopIteration* (página 192) si llega al final de la posición del **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.iteritems()
<dictionary-itemiterator object at 0x7fab9dd4bc58>
>>> for clave,valor in versiones.iteritems():
...     print clave,valor
...
zope 2.13
```

(continué en la próxima página)

(proviene de la página anterior)

```
python 2.7
plone 5.1
>>> versionesIterador = versiones.iteritems()
>>> print versionesIterador.next()
('zope', 2.13)
>>> print versionesIterador.next()
('python', 2.7)
>>> print versionesIterador.next()
('plone', 5.1)
>>> print versionesIterador.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

iterkeys()

Este método devuelve un *iterador* (página 92) sobre las claves del diccionario. Lanza una excepción *StopIteration* (página 192) si llega al final de la posición del **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.iterkeys()
<dictionary-keyiterator object at 0x7fab9dd4bcb0>
>>> for clave in versiones.iterkeys():
...     print clave
...
zope
python
plone
>>> versionesIterador = versiones.iterkeys()
>>> print versionesIterador.next()
zope
>>> print versionesIterador.next()
python
>>> print versionesIterador.next()
plone
>>> print versionesIterador.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

itervalues()

Este método devuelve un *iterador* (página 92) sobre los valores del diccionario. Lanza una excepción *StopIteration* (página 192) si llega al final de la posición del **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.itervalues()
<dictionary-valueiterator object at 0x7fab9dd4bc58>
>>> for valor in versiones.itervalues():
...     print valor
...
2.13
2.7
5.1
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> versionesIterador = versiones.itervalues()
>>> print versionesIterador.next()
2.13
>>> print versionesIterador.next()
2.7
>>> print versionesIterador.next()
5.1
>>> print versionesIterador.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

keys()

Este método devuelve una lista de las claves del diccionario:

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.keys()
['zope', 'python', 'plone']
```

pop()

Este método remueve específicamente una clave de **diccionario** y devuelve valor correspondiente. Lanza una excepción *KeyError* (página 192) si la **clave** no es encontrada.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.pop('zope')
2.13
>>> versiones
{'python': 2.7, 'plone': 5.1}
>>> versiones.pop('django')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'django'
```

popitem()

Este método remueve y devuelve algún par (clave, valor) del **diccionario** como una 2 tuplas. Lanza una excepción *KeyError* (página 192) si el **diccionario** esta vació.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones.popitem()
('zope', 2.13)
>>> versiones
{'python': 2.7, 'plone': 5.1}
>>> versiones.popitem()
('python', 2.7)
>>> versiones
{'plone': 5.1}
>>> versiones.popitem()
('plone', 5.1)
>>> versiones
```

(continué en la próxima página)

(proviene de la página anterior)

```
{}
>>> versiones.popitem()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'popitem(): dictionary is empty'
```

setdefault()

Este método es similar a *get(key, default_value)* (página 68), pero además asigna la clave `key` al valor por `default_value` para la clave si esta no se encuentra en el **diccionario**.

```
D.setdefault(key[,default_value])
```

A continuación un ejemplo de como trabaja el método `setdefault()` cuando la clave esta en el diccionario:

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> zope = versiones.setdefault('zope')
>>> print 'Versiones instaladas:', versiones
Versiones instaladas: {'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> print 'Versión de Zope:', zope
Versión de Zope: 2.13
```

A continuación un ejemplo de como trabaja el método `setdefault()` la clave no esta en el diccionario:

```
>>> paquetes = {'python': 2.7, 'zope': 2.13}
>>> print paquetes
{'python': 2.7, 'zope': 2.13}
>>> plone = paquetes.setdefault('plone')
>>> print 'paquetes: ', paquetes
paquetes:  {'python': 2.7, 'zope': 2.13, 'plone': None}
>>> print 'plone: ', plone
plone:  None
```

Si el valor no es proveído, el valor `default_value` será el tipo objeto integrado *None* (página 210).

A continuación un ejemplo de como trabaja el método `setdefault()` la clave no esta en el diccionario pero esta vez el `default_value` es proveído:

```
>>> pkgs = {'python': 2.7, 'zope': 2.13, 'plone': None}
>>> print pkgs
{'python': 2.7, 'zope': 2.13, 'plone': None}
>>> django = paquetes.setdefault('django', 2.1)
>>> print 'paquetes =', pkgs
paquetes = {'python': 2.7, 'zope': 2.13, 'plone': None}
>>> print 'django =', django
django = 2.1
```

A continuación otro ejemplo en donde puedes agrupar N *tuplas* (página 63) por el valor el cual se repite más y construir un diccionario que cuyas claves son los valores mas repetidos y cuyos valores este agrupados en tipo *listas* (página 58):

```
>>> PKGS = (('zope', 'Zope2'),
...         ('zope', 'pytz'),
...         ('plone', 'Plone'),
...         ('plone', 'diazo'),
...         ('plone', 'z3c.form'),)
>>>
>>> paquetes = {}
>>> for clave, valor in PKGS:
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     if paquetes.has_key(clave):
...         paquetes[clave].append(valor)
...     else:
...         paquetes[clave] = [valor]
...
>>> print paquetes
{'zope': ['Zope2', 'pytz'], 'plone': ['Plone', 'diazo', 'z3c.form']}
```

En el tipo tupla `PKGS` los elementos mas repetidos son `'zope'` y `'plone'` estos se convierten en clave del diccionario `paquetes` y los otros elementos se agrepan en listas como sus respectivos valores.

A continuación un mejor aprovechamiento implementando el método `setdefault()`:

```
>>> PKGS = (('zope', 'Zope2'),
...         ('zope', 'pytz'),
...         ('plone', 'Plone'),
...         ('plone', 'diazo'),
...         ('plone', 'z3c.form'),)
>>> paquetes = {}
>>> for clave, valor in PKGS:
...     paquetes.setdefault(clave, []).append(valor)
...
>>> print paquetes
{'zope': ['Zope2', 'pytz'], 'plone': ['Plone', 'diazo', 'z3c.form']}
```

En el ejemplo anterior puede ver que el aprovechamiento del método `setdefault()` a comparación de no usar el respectivo método.

update()

Este método actualiza un **diccionario** agregando los pares clave-valores en un segundo diccionario. Este método no devuelve nada.

El método `update()` toma un diccionario o un objeto iterable de pares clave/valor (generalmente tuplas). Si se llama a `update()` sin pasar parámetros, el diccionario permanece sin cambios.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
>>> versiones_adicional = dict(django=2.1)
>>> print versiones_adicional
{'django': 2.1}
>>> versiones.update(versiones_adicional)
```

Como puede apreciar este método no devuelve nada, más si muestra de nuevo el diccionario `versiones` puede ver que este fue actualizado con el otro diccionario `versiones_adicional`.

```
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1, 'django': 2.1}
```

values()

Este método devuelve una lista de los valores del diccionario:

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.values()
[2.13, 2.7, 5.1]
```

viewitems()

Este método devuelve un objeto como un conjunto mutable proveyendo una vista en los elementos del diccionario:

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.viewkeys()
dict_keys(['zope', 'python', 'plone'])
>>> for clave,valor in versiones.iteritems():
...     print clave,valor
...
zope 2.13
python 2.7
plone 5.1
```

viewkeys()

Este método devuelve un objeto proveyendo una vista de las claves del **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.viewkeys()
dict_keys(['zope', 'python', 'plone'])
>>> for clave in versiones.viewkeys():
...     print clave
...
zope
python
plone
```

viewvalues()

Este método devuelve un objeto proveyendo una vista de los valores del **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones.viewvalues()
dict_values([2.13, 2.7, 5.1])
>>> for valor in versiones.viewvalues():
...     print valor
...
2.13
2.7
5.1
```

3.11.3 Funciones

Los objetos de tipo **diccionario** tienen disponibles una serie de *funciones* integradas en el interprete Python para su tratamiento, a continuación algunas de estas:

cmp()

Esta función es la misma función integrada *cmp()* (página 125) en el interprete Python pero aplicada al uso de la secuencia de tipo **diccionario**.

```
>>> versiones_proyecto1 = dict(python=2.7, zope=2.13, plone=5.1)
>>> versiones_proyecto2 = dict(django=2.1, django-filter=1.1.0)
>>> print cmp(versiones_proyecto1, versiones_proyecto2)
1
```

La función `cmp()` es usado en Python para comparar valores y claves de dos diccionarios. Si la función devuelve el valor `0` si ambos diccionarios son igual, devuelve el valor `1` si el primer diccionario es mayor que el segundo diccionario y devuelve el valor `-1` si el primer diccionario es menor que el segundo diccionario.

len()

Esta función es la misma función integrada *len()* (página 118) en el interprete Python pero aplicada al uso de la secuencia de tipo **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1)
>>> len(versiones)
3
```

3.11.4 Sentencias

Los objetos de tipo **diccionario** tienen disponibles una serie de *sentencias* integradas en el interprete Python para su tratamiento, a continuación algunas de estas:

del

Esta sentencia es la misma sentencia integrada *del* (página 31) en el interprete Python pero aplicada al uso de la secuencia de tipo **diccionario**.

```
>>> versiones = dict(python=2.7, zope=2.13, plone=5.1, django=2.1)
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1, 'django': 2.1}
>>> del versiones['django']
>>> print versiones
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
```

En el código fuente anterior se usa la sentencia `del` para eliminar un elemento del diccionario mediante su respectiva clave.

3.11.5 Convertir a diccionarios

Para convertir a *tipos diccionarios* debe usar la función *dict()* (página 138) la cual *esta integrada* (página 113) en el interprete Python.

Truco: Para más información consulte las funciones integradas para *operaciones de secuencias* (página 137).

3.11.6 Ejemplos

A continuación, se presentan un ejemplo de su uso:

Ejemplo de definir un diccionario

```
datos_basicos = {
    "nombres":"Leonardo Jose",
    "apellidos":"Caballero Garcia",
    "cedula":"26938401",
    "fecha_nacimiento":"03/12/1980",
    "lugar_nacimiento":"Maracaibo, Zulia, Venezuela",
    "nacionalidad":"Venezolana",
    "estado_civil":"Soltero"
}
```

Ejemplo de operaciones con tipo diccionario con funciones propias

```
print ("\nClaves de diccionario:", datos_basicos.keys())
print ("\nValores de diccionario:", datos_basicos.values())
print ("\nElementos de diccionario:", datos_basicos.items())
```

Ejemplo de iteración avanzada sobre diccionarios con función iteritems

```
for key, value in iter(datos_basicos.items()):
    print('Clave: %s, tiene el valor: %s' % (key, value))
```

Ejemplo real de usar tipo diccionario

```
print ("\n\nInscripción de Curso")
print ("====================")

print ("\nDatos de participante")
print ("----------------------")

print ("Cédula de identidad: ", datos_basicos['cedula'])
print ("Nombre completo: " + datos_basicos['nombres'] + " " + \
datos_basicos['apellidos'])
import datetime, locale, os
locale.setlocale(locale.LC_ALL, os.environ['LANG'])
print ("Fecha y lugar de nacimiento:", datetime.datetime.strftime(
    datetime.datetime.strptime(
        datos_basicos['fecha_nacimiento'], '%d/%m/%Y'
    ), "%d de %B de %Y"
) + " en " + datos_basicos['lugar_nacimiento'] + ".")
print ("Nacionalidad:", datos_basicos['nacionalidad'])
print ("Estado civil:", datos_basicos['estado_civil'])
```

3.11.7 Ayuda integrada

Usted puede consultar toda la documentación disponible sobre los **diccionarios** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(dict)
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `tipo_diccionarios.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python tipo_diccionarios.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

3.12 Tipo conjuntos

Un conjunto, es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas.

Clase	Tipo	Notas	Ejemplo
`set`	Conjuntos	Mutable, sin orden, no contiene duplicados.	`set([4.0, 'Carro', True])`
`frozenset`	Conjuntos	Inmutable, sin orden, no contiene duplicados.	`frozenset([4.0, 'Carro', True])`

3.12.1 Métodos

Los objetos de tipo **conjunto mutable** y **conjunto inmutable** integra una serie de métodos integrados a continuación:

add()

Este método agrega un elemento a un **conjunto mutable**. Esto no tiene efecto si el elemento ya esta presente.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> set_mutable1.add(22)
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11, 22])
```

clear()

Este método remueve todos los elementos desde este **conjunto mutable**.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> set_mutable1.clear()
>>> print set_mutable1
set([])
```

copy()

Este método devuelve una copia superficial del tipo **conjunto mutable** o **conjunto inmutable**:

```
>>> set_mutable = set([4.0, 'Carro', True])
>>> otro_set_mutable = set_mutable.copy()
>>> set_mutable == otro_set_mutable
True
```

difference()

Este método devuelve la diferencia entre dos **conjunto mutable** o **conjunto inmutable**: todos los elementos que están en el primero, pero no en el argumento.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> print set_mutable2
set([2, 4, 5, 8, 9, 11])
>>> print set_mutable1.difference(set_mutable2)
```

(continué en la próxima página)

(proviene de la página anterior)

```
set([1, 3, 7])
>>> print set_mutable2.difference(set_mutable1)
set([8, 9])
```

difference_update()

Este método actualiza un tipo **conjunto mutable** llamando al método `difference_update()` con la diferencia de los conjuntos.

```
>>> proyecto1 = {'python', 'Zope2', 'ZODB3', 'pytz'}
>>> proyecto1
set(['python', 'pytz', 'Zope2', 'ZODB3'])
>>> proyecto2 = {'python', 'Plone', 'diazo'}
>>> proyecto2
set(['python', 'diazo', 'Plone'])
>>> proyecto1.difference_update(proyecto2)
>>> proyecto1
set(['pytz', 'Zope2', 'ZODB3'])
```

Si `proyecto1` y `proyecto2` son dos conjuntos. La diferencia del conjunto `proyecto1` y conjunto `proyecto2` es un conjunto de elementos que existen solamente en el conjunto `proyecto1` pero no en el conjunto `proyecto2`.

discard()

Este método remueve un elemento desde un **conjunto mutable** si esta presente.

```
>>> paquetes = {'python', 'zope', 'plone', 'django'}
>>> paquetes
set(['python', 'zope', 'plone', 'django'])
>>> paquetes.discard('django')
>>> paquetes
set(['python', 'zope', 'plone'])
```

El **conjunto mutable** permanece sin cambio si el elemento pasado como argumento al método `discard()` no existe.

```
>>> paquetes = {'python', 'zope', 'plone', 'django'}
>>> paquetes.discard('php')
>>> paquetes
set(['python', 'zope', 'plone'])
```

intersection()

Este método devuelve la intersección entre los **conjuntos mutables** o **conjuntos inmutables**: todos los elementos que están en ambos.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> print set_mutable2
set([2, 4, 5, 8, 9, 11])
>>> print set_mutable1.intersection(set_mutable2)
set([2, 11, 4, 5])
>>> print set_mutable2.intersection(set_mutable1)
set([2, 11, 4, 5])
```

intersection_update()

Este método actualiza un **conjunto mutable** con la intersección de ese mismo y otro **conjunto mutable**.

El método `intersection_update()` le permite arbitrariamente varios numero de argumentos (conjuntos).

```
>>> proyecto1 = {'python', 'Zope2', 'pytz'}
>>> proyecto1
set(['python', 'pytz', 'Zope2'])
>>> proyecto2 = {'python', 'Plone', 'diazo', 'z3c.form'}
>>> proyecto2
set(['python', 'z3c.form', 'diazo', 'Plone'])
>>> proyecto3 = {'python', 'django', 'django-filter'}
>>> proyecto3
set(['python', 'django-filter', 'django'])
>>> proyecto3.intersection_update(proyecto1, proyecto2)
>>> proyecto3
set(['python'])
```

La intersección de dos o mas conjuntos es el conjunto de elemento el cual es común a todos los conjuntos.

isdisjoint()

Este método devuelve el valor `True` si no hay elementos comunes entre los **conjuntos mutables** o **conjuntos inmutables**.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> print set_mutable2
set([2, 4, 5, 8, 9, 11])
>>> print set_mutable1.isdisjoint(set_mutable2)
```

issubset()

Este método devuelve el valor `True` si el **conjunto mutable** es un *subconjunto* del **conjunto mutable** o del **conjunto inmutable** argumento.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> set_mutable3 = set([11, 5, 2, 4])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> print set_mutable2
set([2, 4, 5, 8, 9, 11])
>>> print set_mutable3
set([2, 11, 4, 5])
>>> print set_mutable2.issubset(set_mutable1)
False
>>> print set_mutable3.issubset(set_mutable1)
True
```

issuperset()

Este método devuelve el valor `True` si el **conjunto mutable** o el **conjunto inmutable** es un *superset* del **conjunto mutable** argumento.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> set_mutable3 = set([11, 5, 2, 4])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> print set_mutable2
set([2, 4, 5, 8, 9, 11])
>>> print set_mutable3
set([2, 11, 4, 5])
>>> print set_mutable1.issuperset(set_mutable2)
False
>>> print set_mutable1.issuperset(set_mutable3)
True
```

pop()

Este método remueve arbitrariamente y devuelve un elemento de **conjunto mutable**. El método `pop()` no toma ningún argumento. Si el **conjunto mutable** esta vacío se lanza una excepción *KeyError* (página 192).

```
>>> paquetes = {'python', 'zope', 'plone', 'django'}
>>> paquetes
set(['python', 'zope', 'plone', 'django'])
>>> print "Valor aleatorio devuelto es:", paquetes.pop()
Valor aleatorio devuelto es: python
>>> paquetes
set(['zope', 'plone', 'django'])
>>> print "Valor aleatorio devuelto es:", paquetes.pop()
Valor aleatorio devuelto es: zope
>>> paquetes
set(['plone', 'django'])
>>> print "Valor aleatorio devuelto es:", paquetes.pop()
Valor aleatorio devuelto es: plone
>>> paquetes
set(['django'])
>>> print "Valor aleatorio devuelto es:", paquetes.pop()
Valor aleatorio devuelto es: django
>>> print "Valor aleatorio devuelto es:", paquetes.pop()
Valor aleatorio devuelto es:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

Tenga en cuenta que usted podría obtener diferente salida devueltas usando el método `pop()` por que remueve aleatoriamente un elemento.

remove()

Este método busca y remueve un elemento de un **conjunto mutable**, si debe ser un miembro.

```
>>> paquetes = {'python', 'zope', 'plone', 'django'}
>>> paquetes
set(['python', 'zope', 'plone', 'django'])
>>> paquetes.remove('django')
>>> paquetes
set(['python', 'zope', 'plone'])
```

Si el elemento no es existe en el **conjunto mutable**, lanza una excepción *KeyError* (página 192). Usted puede usar el método *discard()* (página 77) si usted no quiere este error. El **conjunto mutable** permanece sin cambio si el elemento pasado al método `discard()` no existe.

Un conjunto es una colección desordenada de elementos. Si usted quiere remover arbitrariamente elemento un conjunto, usted puede usar el método *pop()* (página 79).

symmetric_difference()

Este método devuelve todos los elementos que están en un **conjunto mutable** e **conjunto inmutable** u otro, pero no en ambos.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> print set_mutable2
set([2, 4, 5, 8, 9, 11])
>>> print set_mutable1.symmetric_difference(set_mutable2)
set([1, 3, 7, 8, 9])
```

symmetric_difference_update()

Este método actualiza un **conjunto mutable** llamando al método `symmetric_difference_update()` con los conjuntos de diferencia simétrica.

La diferencia simétrica de dos conjuntos es el conjunto de elementos que están en cualquiera de los conjuntos pero no en ambos.

```
>>> proyecto1 = {'python', 'plone', 'django'}
>>> proyecto1
set(['python', 'plone', 'django'])
>>> proyecto2 = {'django', 'zope', 'pyramid'}
>>> proyecto2
set(['zope', 'pyramid', 'django'])
>>> proyecto1.symmetric_difference_update(proyecto2)
>>> proyecto1
set(['python', 'zope', 'pyramid', 'plone'])
```

El método `symmetric_difference_update()` toma un argumento simple de un tipo **conjunto mutable**.

union()

Este método devuelve un **conjunto mutable** y **conjunto inmutable** con todos los elementos que están en alguno de los **conjuntos mutables** y **conjuntos inmutables**.

```
>>> set_mutable1 = set([4, 3, 11, 7, 5, 2, 1, 4])
>>> set_mutable2 = set([11, 5, 9, 2, 4, 8])
>>> print set_mutable1
set([1, 2, 3, 4, 5, 7, 11])
>>> print set_mutable2
set([2, 4, 5, 8, 9, 11])
>>> print set_mutable1.union(set_mutable2)
set([1, 2, 3, 4, 5, 7, 8, 9, 11])
```

update()

Este método agrega elementos desde un **conjunto mutable** (pasando como un argumento) un tipo *tupla* (página 63), un tipo *lista* (página 58), un tipo *diccionario* (página 65) o un tipo **conjunto mutable** llamado con el método `update()`.

A continuación un ejemplo de agregar nuevos elementos un tipo **conjunto mutable** usando otro tipo **conjunto mutable**:

```
>>> version_plone_dev = set([5.1, 6])
>>> version_plone_dev
set([5.1, 6])
>>> versiones_plone = set([2.1, 2.5, 3.6, 4])
>>> versiones_plone
set([2.5, 3.6, 2.1, 4])
>>> versiones_plone.update(version_plone_dev)
>>> versiones_plone
set([2.5, 3.6, 4, 6, 5.1, 2.1])
```

A continuación un ejemplo de agregar nuevos elementos un tipo **conjunto mutable** usando otro tipo *cadena de caracteres* (página 46):

```
>>> cadena = 'abc'
>>> cadena
'abc'
>>> conjunto = {1, 2}
>>> conjunto.update(cadena)
>>> conjunto
set(['a', 1, 2, 'b', 'c'])
```

A continuación un ejemplo de agregar nuevos elementos un tipo **conjunto mutable** usando otro tipo *diccionario* (página 65):

```
>>> diccionario = {'key': 1, 2:'lock'}
>>> diccionario.viewitems()
dict_items([(2, 'lock'), ('key', 1)])
>>> conjunto = {'a', 'b'}
>>> conjunto.update(diccionario)
>>> conjunto
set(['a', 2, 'b', 'key'])
```

3.12.2 Convertir a conjuntos

Para convertir a *tipos conjuntos* debe usar las funciones *set()* (página 140) y *frozenset()* (página 138), las cuales *están integradas* (página 113) en el interprete Python.

Truco: Para más información consulte las funciones integradas para *operaciones de secuencias* (página 137).

3.12.3 Ejemplos

Conjuntos set

A continuación, se presentan un ejemplo de conjuntos `set`:

```
# crea un conjunto sin valores repetidos y lo asigna la variable
para_comer = set([
        'pastel', 'tequeno', 'papa', 'empanada', 'mandoca'])
print (para_comer, type(para_comer))
para_tomar = set(['refresco', 'malta', 'jugo', 'cafe'])
print (para_tomar, type(para_tomar))

# usa operaciones condicionales con operador in
hay_tequeno = 'tequeno' in para_comer
```

(continué en la próxima página)

(proviene de la página anterior)

```
hay_fresco = 'refresco' in para_tomar

print ("\nTostadas A que Pipo!")
print ("====================")

# valida si un elemento esta en el conjunto
print ("Tenéis tequeno?:", 'tequeno' in para_comer)

# valida si un elemento esta en el conjunto
print ("Tenéis pa' tomar fresco?:", 'refresco' in para_tomar)

if (hay_tequeno and hay_fresco):
    print ("Desayuno vergatario!!!")
else:
    print ("Desayuno ligero")
```

Conjuntos frozenset

A continuación, se presentan un ejemplo de conjuntos `frozenset`:

```
>>> versiones_plone = frozenset([6, 2.1, 2.5, 3.6, 4, 5, 4, 2.5])
>>> print versiones_plone, type(versiones_plone)
frozenset([2.5, 4, 5, 6, 2.1, 3.6]) <type 'frozenset'>
```

Los elementos de un set son únicos (sin repeticiones dentro del `set`), y deben ser objetos inmutables: números, cadenas de caracteres, tuplas y sets inmutables, pero no listas ni sets mutables.

3.12.4 Ayuda integrada

Usted puede consultar toda la documentación disponible sobre los **conjuntos set** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(set)
```

Usted puede consultar toda la documentación disponible sobre los **conjuntos frozenset** desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(frozenset)
```

Para salir de esa ayuda presione la tecla `q`.

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `tipo_conjuntos.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python tipo_conjuntos.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 262) del entrenamiento para ampliar su conocimiento en esta temática.

CAPÍTULO 4

Bloques de código y estructuras de control

En Python tiene las estructuras de control como `if` (`elif`, `else`); `for`, `while` (`else`, `break`, `continue`, `pass`); las funciones *range()* (página 120) y *xrange()* (página 121); ademas de los tipos *iteradores* (página 92).

En esta lección se describen las estructuras de control del lenguaje Python, mostrando ejemplos prácticos y útiles. A continuación el temario de esta lección:

4.1 Condicional if

La sentencia condicional `if` se usa para tomar decisiones, este evaluá básicamente una operación lógica, es decir una expresión que de como resultado `True` o `False`, y ejecuta la pieza de código siguiente siempre y cuando el resultado sea verdadero.

A continuación un de estructura condicional `if`/`elif`/`else` completo:

```
numero = int(input("\nIngresa un número entero, por favor: "))

if numero < 0:
    numero = 0
    print ('El número ingresado es negativo cambiado a cero.\n')
elif numero == 0:
    print ('El número ingresado es 0.\n')
elif numero == 1:
    print ('El número ingresado es 1.\n')
else:
    print ('El número ingresado es mayor que uno.\n')
```

En el ejemplo anterior usa dos funciones integradas en el interprete Python:

- La función *int()* (página 126) que convierte el valor ingresado a número entero.
- La función *raw_input()* (página 123) lee el valor ingresado por la entrada estándar.

El valor es ingresado en la variable `numero` comprobará en el sentencia condicional `if`, si la comprobación devuelve `False` intentará con el siguiente bloque condicional `elif`, si la comprobación devuelve `False` nuevamente intentará con el siguiente bloque condicional `elif` si de nuevo la comprobación devuelve `False` por ultimo intentará con el siguiente bloque condicional `else` la cual se ejecutara sin comprobación.

4.1.1 Sentencia if

La sentencia `if EXPRESION`, significa, **Si** se cumple la *expresión condicional* (página 84) se ejecuta el bloque de sentencias seguidas.

4.1.2 Sentencia elif

La sentencia `elif EXPRESION`, significa, **De lo contrario Si** se cumple la *expresión condicional* (página 84) se ejecuta el bloque de sentencias seguidas.

4.1.3 Sentencia else

La sentencia `else`, significa, **De lo contrario** se cumple sin evaluar ninguna *expresión condicional* (página 84) y ejecuta el bloque de sentencias seguidas.

4.1.4 Operador is

El operador `is`, significa, que prueba identidad: ambos lados de la expresión condicional debe ser el mismo objecto:

```
>>> 1 is 1.
False
>>> a, b = 1, 1
>>> a is b
True
```

4.1.5 Operador in

El operador `in`, significa, para cualquier colección del valor del lado izquierdo contenga el valor del lado derecho:

```
>>> b = [1, 2, 3]
>>> 2 in b
True
>>> 5 in b
False
```

En el ejemplo anterior, si `b` es una lista, este prueba que `2` y `5` sean elementos de la lista `b`.

4.1.6 Operador not in

El operador `not in`, el contrario de operador *in* (página 84), devuelve `True` cuando un elemento no está en una secuencia.

```
>>> b = [1, 2, 3]
>>> 4 not in b
True
>>> 1 not in b
False
```

En el ejemplo anterior, si `b` es una lista, este prueba que `4` y `1` sean elementos de la lista `b`.

4.1.7 Expresiones condicional

Estos son los distintos tipos de expresiones condicionales:

Expresión if

La expresión de la sentencia `if` se evaluá a `False` cuando se cumple las siguientes expresiones están presente:

- Cualquier numero igual a cero (0, 0.0, 0+0j).
- Un contenedor vació (*lista* (página 58), *tupla* (página 63), *conjunto* (página 75), *diccionario* (página 65)).
- `False`, `None`.

De lo contrario evaluá a `True` cuando se cumple la siguiente expresión esta presente:

- cualquier cosa de lo contrario.

```
if EXPRESION:
    pass
```

Expresión ==

Esta expresión usa el operador `==` (página 38) para validar la misma.

Expresión is

Esta expresión usa el operador *is* (página 84) para validar la misma.

Expresión in

Esta expresión usa el operador *in* (página 84) para validar la misma.

4.1.8 Ejemplos

A continuación, se presenta un ejemplo del uso de condicionales `if`:

Definir variables usadas en los siguientes ejemplos:

```
dato1, dato2, dato3, dato4 = 21, 10, 5, 20;
```

Ejemplo de operador de comparación Igual:

```
if (dato1 == dato2):
    print ("'dato1' y 'dato2' son iguales.")
else:
    print ("'dato1' y 'dato2' no son iguales.")
```

Ejemplo de operador de comparación Distinto:

```
if (dato1 != dato2):
    print ("'dato1' y 'dato2' son distintas/diferentes.")
else:
    print ("'dato1' y 'dato2' no son distintas/diferentes.")
```

Ejemplo de operador de comparación Diferente:

```
if (dato1 != dato2):
    print ("'dato1' y 'dato2' son distintas/diferentes.")
else:
    print ("'dato1' y 'dato2' no son distintas/diferentes.")
```

Ejemplo de operador de comparación Menor que:

```
if (dato1 < dato2):
    print ("'dato1' es menor que 'dato2'.")
else:
    print ("'dato1' no es menor que 'dato2'.")
```

Ejemplo de operador de comparación Mayor que:

```
if (dato1 > dato2):
    print ("'dato1' es mayor que 'dato2'.")
else:
    print ("'dato1' no es mayor que 'dato2'.")
```

Ejemplo de operador de comparación Menor o igual que:

```
if (dato3 <= dato4):
    print ("'dato3' es menor o igual que 'dato4'.")
else:
    print ("'dato3' no es menor o igual que 'dato4'.")
```

Ejemplo de operador de comparación Mayor o igual que:

```
if (dato4 >= dato3):
    print ("'dato4' es mayor o igual que 'dato3'.")
else:
    print ("'dato4' no es mayor o igual que 'dato3'.")
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `condicional_if.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python condicional_if.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 263) del entrenamiento para ampliar su conocimiento en esta temática.

4.2 Operadores lógicos

Estos son los distintos tipos de operadores con los que puede trabajar con valores booleanos, los llamados operadores lógicos o condicionales:

4.2.1 Operador and

El operador `and` evalúa si el valor del lado izquierdo y el lado derecho *se cumple*.

```
>>> True and False
False
```

4.2.2 Operador or

El operador `or` evalúa si el valor del lado izquierdo o el lado derecho *se cumple*.

```
>>> True or False
True
```

4.2.3 Operador not

El operador `not` devuelve el valor *opuesto* la valor booleano.

```
>>> not True
False
```

Si la expresión es `True` el valor devuelto es `False`, de lo contrario si la expresión es `False` el valor devuelto es `True`.

```
>>> not False
True
```

4.2.4 Ejemplos

A continuación, se presentan algunos ejemplos de su uso:

Definir variables usadas en los siguientes ejemplos:

```
a, b = 10, 20
```

Ejemplo de operador lógico and:

```
if (a and b):
   print ("Las variables 'a' y 'b' son VERDADERO.")
else:
   print ("O bien la variable 'a' no es VERDADERO " + \
   "o la variable 'b' no es VERDADERO.")
```

Ejemplo de operador lógico or:

```
if (a or b):
   print ("O bien la variable 'a' es VERDADERA " + \
   "o la variable 'b' es VERDADERA " + \
   "o ambas variables son VERDADERAS.")
else:
   print ("Ni la variable 'a' es VERDADERA ni " + \
   "la variable 'b' es VERDADERA.")
```

Ejemplo de operador lógico not:

```
if not(a and b):
   print ("Ni la variable 'a' NO es VERDADERA " + \
   "o la variable 'b' NO es VERDADERA.")
else:
   print ("Las variables 'a' y 'b' son VERDADERAS.")
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `operadores_logicos.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python operadores_logicos.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 263) del entrenamiento para ampliar su conocimiento en esta temática.

4.3 Bucle while

En Python tiene una palabra reservada llamada `while` que nos permite ejecutar ciclos, o bien secuencias periódicas que nos permiten ejecutar código múltiples veces.

El ciclo `while` nos permite realizar múltiples iteraciones basándonos en el resultado de una expresión lógica que puede tener como resultado un valor `True` o `False`.

4.3.1 Tipos de Bucle "while"

A continuación, se presentan algunos ejemplos del uso del bucle `while`:

Bucle "while" controlado por Conteo

A continuación, se presenta un ejemplo del uso del bucle `while` controlado por conteo:

```
suma, numero = 0, 1

while numero <= 10:
    suma = numero + suma
    numero = numero + 1
print ("La suma es " + str(suma))
```

En este ejemplo tiene un contador con un valor inicial de cero, cada iteración del `while` manipula esta variable de manera que incremente su valor en 1, por lo que después de su primera iteración el contador tendrá un valor de 1, luego 2, y así sucesivamente.

Eventualmente cuando el contador llegue a tener un valor de 10, la condición del ciclo `numero <= 10` sera `False`, por lo que el ciclo terminará arrojando el siguiente resultado.

Bucle "while" controlado por Evento

A continuación, se presenta un ejemplo del uso del bucle `while` controlado por Evento:

```
promedio, total, contar = 0.0, 0, 0

print ("Introduzca la nota de un estudiante (-1 para salir): ")
grado = int(input())
while grado != -1:
    total = total + grado
    contar = contar + 1
    print ("Introduzca la nota de un estudiante (-1 para salir): ")
    grado = int(input())
```

(continué en la próxima página)

(proviene de la página anterior)

```
promedio = total / contar
print ("Promedio de notas del grado escolar es: " + str(promedio))
```

En este casi el evento que se dispara cuando el usuario ingresa el valor `-1`, causando que el bucle `while` se interrumpo o no se inicie.

Bucle "while" con "else"

Al igual que la sentencia *if* (página 84), la estructura `while` también puede combinarse con una sentencia *else* (página 84)).

El nombre de la sentencia *else* (página 84) es equivocada, ya que el bloque `else` se ejecutará en todos los casos, es decir, cuando la expresión condicional del `while` sea `False`, (a comparación de la *sentencia if* (página 84)).

```
promedio, total, contar = 0.0, 0, 0
mensaje = "Introduzca la nota de un estudiante (-1 para salir): "

grado = int(input(mensaje))
while grado != -1:
    total = total + grado
    contar += 1
    grado = int(input(mensaje))
else:
    promedio = total / contar
    print ("Promedio de notas del grado escolar: " + str(promedio))
```

La sentencia `else` tiene la ventaja de mantener el mismo nombre y la misma sintaxis que en las demás estructuras de control.

4.3.2 Sentencias utilitarias

A continuación, se presentan algunos ejemplos del uso de sentencias utilitarias usadas en el bucle `while`:

Sentencia break

A continuación, se presenta un ejemplo del uso del bucle `while` controlado la sentencia `break`:

```
variable = 10

while variable > 0:
    print ("Actual valor de variable:", variable)
    variable = variable -1
    if variable == 5:
        break
```

Adicionalmente existe una forma alternativa de interrumpir o cortar los ciclos utilizando la palabra reservada `break`.

Esta nos permite salir del ciclo incluso si la expresión evaluada en `while` (o en otro ciclo como `for`) permanece siendo `True`. Para comprender mejor use el mismo ejemplo anterior pero se interrumpe el ciclo usando la sentencia `break`.

Sentencia continue

A continuación, se presenta un ejemplo del uso del bucle `while` controlado la sentencia `continue`:

```
variable = 10

while variable > 0:
   variable = variable -1
   if variable == 5:
      continue
   print ("Actual valor de variable:", variable)
```

La sentencia `continue` hace que pase de nuevo al principio del bucle aunque no se haya terminado de ejecutar el ciclo anterior.

4.3.3 Ejemplos

Sucesión de Fibonacci

Ejemplo de la Sucesión de Fibonacci[59] con bucle `while`:

```
a, b = 0, 1
while b < 100:
    print (b,)
    a, b = b, a + b
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `bucle_while.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python bucle_while.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 263) del entrenamiento para ampliar su conocimiento en esta temática.

4.4 Bucle for

La sentencia `for` en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia *for* de Python itera sobre los ítems de cualquier secuencia (una lista o una cadenas de caracteres), en el orden que aparecen en la secuencia.

4.4.1 Tipos de Bucle "for"

A continuación, se presentan algunos ejemplos del uso del bucle `for`:

[59] https://es.wikipedia.org/wiki/Sucesión_de_Fibonacci

Bucle "for" con Listas

A continuación, se presenta un ejemplo del uso del bucle `for` con tipos de estructuras de datos *listas* (página 58):

```
animales = ['gato', 'perro', 'serpiente']
for animal in animales:
    print ("El animal es: {0}, tamaño de palabra es: {1}".format(
        animal, len(animal)))
```

Bucle "for" con Listas y función "range"

A continuación, se presenta un ejemplo del uso del bucle `for` con tipos de estructuras de datos *listas* (página 58) con la función *range()* (página 120) y la función *len()* (página 118):

```
oracion = 'Mary entiende muy bien Python'
frases = oracion.split() # convierte a una lista cada palabra
print ("La oración analizada es:", oracion, ".\n")
for palabra in range(len(frases)):
    print ("Palabra: {0}, en la frase su posición es: {1}".format(
        frases[palabra], palabra))
```

Si se necesita iterar sobre una secuencia de números. Genera una lista conteniendo progresiones aritméticos, por ejemplo, como se hace en el fragmento de código fuente anterior.

Bucle "for" con Tuplas

A continuación, se presenta un ejemplo del uso del bucle `for` con tipos de estructuras de datos *Tuplas* (página 63):

```
conexion_bd = "127.0.0.1","root","123456","nomina"
for parametro in conexion_bd:
    print (parametro)
```

El ejemplo anterior itera una *tupla* (página 63) de parámetros.

Bucle "for" con Diccionarios

A continuación, se presenta un ejemplo del uso del bucle `for` con tipos de estructuras de datos *diccionarios* (página 65):

```
datos_basicos = {
    "nombres":"Leonardo Jose",
    "apellidos":"Caballero Garcia",
    "cedula":"26938401",
    "fecha_nacimiento":"03/12/1980",
    "lugar_nacimiento":"Maracaibo, Zulia, Venezuela",
    "nacionalidad":"Venezolana",
    "estado_civil":"Soltero"
}
clave = datos_basicos.keys()
valor = datos_basicos.values()
cantidad_datos = datos_basicos.items()

for clave, valor in cantidad_datos:
    print (clave + ": " + valor)
```

El ejemplo anterior itera un *diccionario* (página 65) con datos básicos de una persona.

Bucle "for" con "else"

Al igual que la sentencia *if* (página 83) y el bucle *while* (página 88), la estructura `for` también puede combinarse con una sentencia *else* (página 84).

El nombre de la sentencia *else* (página 84) es equivocada, ya que el bloque *else* (página 84) se ejecutará en todos los casos, es decir, cuando la expresión condicional del bucle `for` sea `False`, (a comparación de la *sentencia if* (página 83)).

```
db_connection = "127.0.0.1","5432","root","nomina"
for parametro in db_connection:
    print (parametro)
else:
    print ("""El comando PostgreSQL es:
$ psql -h {server} -p {port} -U {user} -d {db_name}""".format(
        server=db_connection[0], port=db_connection[1],
        user=db_connection[2], db_name=db_connection[3]))
```

La sentencia `else` tiene la ventaja de mantener el mismo nombre y la misma sintaxis que en las demás estructuras de control.

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `bucle_for.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python bucle_for.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 263) del entrenamiento para ampliar su conocimiento en esta temática.

4.5 Iteradores

Para entender los la filosofía de los *Iteradores*, busca la *simplicidad* de las operaciones, evitando la duplicación del esfuerzo, el cual es un derroche y busca reemplazar varios de los enfoques propios con una característica estándar, normalmente, deriva en hacer las cosas más legibles además de más interoperable.

> *Guido van Rossum — Añadiendo tipado estático opcional a Python* (Adding Optional Static Typing to Python[60]).

Un iterador es un objeto adherido al iterator protocol[61], básicamente esto significa que tiene una función *next()* (página 139), es decir, cuando se le llama, devuelve la siguiente elemento en la secuencia, cuando no queda nada para ser devuelto, lanza la excepción *StopIteration* (página 192) y se causa el detener la iteración. Pero si se llama de forma explícita puede ver que, una vez que el iterador esté *agotado*, al llamarlo nuevamente verá que se lanza la excepción comentada anteriormente.

A continuación el uso de iteradores usando del método especial `__iter__()` incluido en el *objeto integrado file* (página 214):

[60] https://www.artima.com/weblogs/viewpost.jsp?thread=86641
[61] https://docs.python.org/dev/library/stdtypes.html#iterator-types

```
>>> archivo = open('/etc/hostname')
>>> archivo
<open file '/etc/hostname', mode 'r' at 0x7fa44ba379c0>
>>> archivo.__iter__()
<open file '/etc/hostname', mode 'r' at 0x7fa44ba379c0>
>>> iter(archivo)
<open file '/etc/hostname', mode 'r' at 0x7fa44ba379c0>
>>> archivo is archivo.__iter__()
True
>>> linea = archivo.__iter__()
>>> linea.next()
'laptop\n'
>>> linea.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo, el método especial `__iter__()`, hace lo mismo que la función integrada *iter(archivo)* (página 138).

4.5.1 Iteradores y secuencias

Los *iteradores* se usan con los tipos de secuencias estándar. A continuación, se describen algunos ejemplos:

Iterar sobre la secuencia inmutable cadena de carácter

A continuación, un ejemplo del uso de los iteradores con la secuencia *inmutable* de tipo *cadena de caracteres* (página 46) `ASCII`:

```
>>> frase = 'Hola Mundo'
>>> letra = iter(frase)
>>> letra.next()
'H'
>>> letra.next()
'o'
>>> letra.next()
'l'
>>> letra.next()
'a'
>>> letra.next()
' '
>>> letra.next()
'M'
>>> letra.next()
'u'
>>> letra.next()
'n'
>>> letra.next()
'd'
>>> letra.next()
'o'
>>> letra.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `frase`, al llegar al final mediante el iterador `letra` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Iterar sobre la secuencia inmutable cadena Unicode

A continuación, un ejemplo del uso de los iteradores con la secuencia *inmutable* de tipo *cadena de caracteres* (página 47) `Unicode`:

```
>>> frase = u'Jekechitü'
>>> letra = iter(frase)
>>> letra.next()
u'J'
>>> letra.next()
u'e'
>>> letra.next()
u'k'
>>> letra.next()
u'e'
>>> letra.next()
u'c'
>>> letra.next()
u'h'
>>> letra.next()
u'i'
>>> letra.next()
u't'
>>> letra.next()
u'\xfc'
>>> letra.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `frase`, al llegar al final mediante el iterador `letra` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Iterar sobre la secuencia inmutable tupla

A continuación, un ejemplo del uso de los iteradores con la secuencia *inmutable* de tipo *tupla* (página 63):

```
>>> valores = ("Python", True, "Zope", 5)
>>> valores
('Python', True, "Zope", 5)
>>> valores.__iter__()
<tupleiterator object at 0x7fa44b9fa450>
>>> valor = valores.__iter__()
>>> valor.next()
'Python'
>>> valor.next()
True
>>> valor.next()
'Zope'
>>> valor.next()
5
>>> valor.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `valores`, al llegar al final mediante el iterador `valor` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Iterar sobre la función inmutable xrange

A continuación, un ejemplo del uso de los iteradores con la secuencia *inmutable* con la función integrada *xrange()* (página 121):

```
>>> lista = iter(xrange(5))
>>> lista
```

(continué en la próxima página)

(proviene de la página anterior)

```
<rangeiterator object at 0x7fa44b9fb7b0>
>>> lista.next()
0
>>> lista.next()
1
>>> lista.next()
2
>>> lista.next()
3
>>> lista.next()
4
>>> lista.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `lista`, al llegar al final se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Iterar sobre la secuencia mutable lista

A continuación, un ejemplo del uso de los iteradores con la secuencia *mutable* de tipo *lista* (página 58):

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> iter(versiones_plone)
<listiterator object at 0x7fa44b9fa450>
>>> version = iter(versiones_plone)
>>> version
<listiterator object at 0x7fa44b9fa550>
>>> version.next()
2.1
>>> version.next()
2.5
>>> version.next()
3.6
>>> version.next()
4
>>> version.next()
5
>>> version.next()
6
>>> version.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `versiones_plone`, al llegar al final mediante el iterador `version` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Usted puede devolver un objeto iterador en orden inverso sobre una secuencia *mutable* de tipo *lista* (página 58) usando su función integrada `__reversed__()`.

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> versiones_plone.__reversed__()
<listreverseiterator object at 0xb712ebec>
>>> version = versiones_plone.__reversed__()
>>> version.next()
6
>>> version.next()
5
>>> version.next()
4
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> version.next()
3.6
>>> version.next()
2.5
>>> version.next()
2.1
>>> version.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `versiones_plone`, al llegar al final mediante el iterador `version` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

También puede acceder al uso del método especial `__iter__()` incluido en la secuencia *mutable* del tipo integrado *lista* (página 58):

```
>>> versiones_plone = [2.1, 2.5, 3.6, 4, 5, 6]
>>> versiones_plone.__iter__()
<listiterator object at 0x7fa44b9fa510>
```

Iterar sobre la función mutable range

A continuación, un ejemplo del uso de los iteradores con la secuencia *mutable* de la función integrada *range()* (página 120):

```
>>> lista = iter(range(5))
>>> lista
<listiterator object at 0x7fa44b9fa490>
>>> lista.next()
0
>>> lista.next()
1
>>> lista.next()
2
>>> lista.next()
3
>>> lista.next()
4
>>> lista.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `lista`, al llegar al final se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

4.5.2 Iteradores y conjuntos

Los *iteradores* se usan con los tipos de conjuntos estándar. A continuación, se describen algunos ejemplos:

Iterar sobre el conjunto mutable

A continuación, un ejemplo del uso de los iteradores con el conjunto *mutable* de tipo *conjuntos* (página 75):

```
>>> versiones_plone = set([2.1, 2.5, 3.6, 4, 5, 6, 4])
>>> version = iter(versiones_plone)
>>> version
<setiterator object at 0x7fac9c7c7a50>
>>> version.next()
2.5
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> version.next()
4
>>> version.next()
5
>>> version.next()
6
>>> version.next()
2.1
>>> version.next()
3.6
>>> version.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `versiones_plone`, al llegar al final mediante el iterador `version` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Iterar sobre el conjunto inmutable

A continuación, un ejemplo del uso de los iteradores con el conjunto *inmutable* de tipo *conjuntos* (página 75):

```
>>> versiones_plone = frozenset([6, 2.1, 2.5, 3.6, 4, 5, 4, 2.5])
>>> version = iter(versiones_plone)
>>> version
<setiterator object at 0x7fac9c7c7cd0>
>>> version.next()
2.5
>>> version.next()
4
>>> version.next()
5
>>> version.next()
6
>>> version.next()
2.1
>>> version.next()
3.6
>>> version.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `versiones_plone`, al llegar al final mediante el iterador `version` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

4.5.3 Iteradores y mapeos

Los *iteradores* se usan con los tipos de secuencias estándar. A continuación, se describen algunos ejemplos:

Iterar sobre las claves del diccionario

A continuación, un ejemplo del uso de los iteradores con la secuencia de *mapeo*, tipo *diccionario* (página 65), por defecto muestra la clave de la secuencia:

```
>>> versiones_plone = dict(python=2.7, zope=2.13, plone=5.1)
>>> paquete = iter(versiones_plone)
>>> paquete
<dictionary-keyiterator object at 0x7fa44b9e99f0>
>>> paquete.next()
'zope'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> paquete.next()
'python'
>>> paquete.next()
'plone'
>>> paquete.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `versiones_plone`, al llegar al final mediante el iterador `paquete` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Iterar sobre los valores del diccionario

A continuación, un ejemplo del uso de los iteradores con la secuencia de *mapeo*, tipo *diccionario* (página 65) para mostrar el valor de una clave usando el método integrado *itervalues()* (página 69):

```
>>> versiones_plone = dict(python=2.7, zope=2.13, plone=5.1)
>>> version = iter(versiones_plone.itervalues())
>>> version
<dictionary-valueiterator object at 0x7fa44b9e9c00>
>>> version.next()
2.13
>>> version.next()
2.7
>>> version.next()
5.1
>>> version.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `versiones_plone`, al llegar al final mediante el iterador `version` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Iterar sobre los elementos del diccionario

A continuación, un ejemplo del uso de los iteradores con la secuencia de *mapeo*, tipo *diccionario* (página 65) para mostrar el par clave/valor usando el método integrado *iteritems()* (página 68):

```
>>> versiones_plone = dict(python=2.7, zope=2.13, plone=5.1)
>>> paquete = iter(versiones_plone.iteritems())
>>> paquete
<dictionary-itemiterator object at 0x7fa44b9e9b50>
>>> paquete.next()
('zope', 2.13)
>>> paquete.next()
('python', 2.7)
>>> paquete.next()
('plone', 5.1)
>>> paquete.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia `versiones_plone`, al llegar al final mediante el iterador `paquete` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

Ver también:

Consulte la sección de *lecturas suplementarias* (página 266) del entrenamiento para ampliar su conocimiento en esta temática.

CAPÍTULO 5

Funciones y programación estructurada

En Python tiene las estructuras de `functions` los cuales son bloques de código fuente que pueden contener sentencias reusables de código fuente que puede ser personalizables vía parámetros.

En esta lección se describen las estructuras de funciones del lenguaje Python, mostrando ejemplos prácticos y útiles. A continuación el temario de esta lección:

5.1 Programación estructurada

La programación estructurada es un paradigma de programación basado en utilizar *funciones* (página 100) o subrutinas, y únicamente tres estructuras de control:

- secuencia: ejecución de una sentencia tras otra.
- selección o condicional: ejecución de una sentencia o conjunto de sentencias, según el valor de una variable booleana.
- iteración (ciclo o bucle): ejecución de una sentencia o conjunto de sentencias, mientras una variable booleana sea verdadera.

Este paradigma se fundamente en el teorema correspondiente, que establece que toda función computable puede ser implementada en un lenguaje de programación que combine sólo estas tres estructuras lógicas o de control.

La estructura de secuencia es la que se da naturalmente en el lenguaje, ya que por defecto las sentencias son ejecutadas en el orden en que aparecen escritas en el programa.

Para las estructuras condicionales o de selección, Python dispone de la sentencia *if* (página 84), que puede combinarse con sentencias *elif* (página 84) y/o *else* (página 84).

Para los bucles o iteraciones existen las estructuras *while* (página 88) y *for* (página 90).

5.1.1 Ventajas del paradigma

Entre las ventajas de la programación estructurada sobre el modelo anterior (hoy llamado despectivamente *código espagueti*), cabe citar las siguientes:

- Los programas son más fáciles de entender, pueden ser leídos de forma secuencial y no hay necesidad de tener que rastrear saltos de líneas (GOTO) dentro de los bloques de código para intentar entender la lógica interna.

- La estructura de los programas es clara, puesto que las sentencias están más ligadas o relacionadas entre sí.
- Se optimiza el esfuerzo en las fases de pruebas y depuración. El seguimiento de los fallos o errores del programa (debugging), y con él su detección y corrección, se facilita enormemente.
- Se reducen los costos de mantenimiento. Análogamente a la depuración, durante la fase de mantenimiento, modificar o extender los programas resulta más fácil.
- Los programas son más sencillos y más rápidos de confeccionar.
- Se incrementa el rendimiento de los programadores.

Importante: Seguidamente se presenta el concepto de funciones poniendo en practica forma de secuencia, implementado las estructuras condicionales *if* (página 83) y los bucles o iteraciones existen *while* (página 88) y *for* (página 90).

Ver también:

Consulte la sección de *lecturas suplementarias* (página 264) del entrenamiento para ampliar su conocimiento en esta temática.

5.2 Funciones

Una función es un bloque de código con un nombre asociado, que recibe cero o más argumentos como entrada, sigue una secuencia de sentencias, la cuales ejecuta una operación deseada y devuelve un valor y/o realiza una tarea, este bloque puede ser llamados cuando se necesite.

El uso de funciones es un componente muy importante del paradigma de la programación llamada *estructurada* (página 99), y tiene varias ventajas:

- **modularización**: permite segmentar un programa complejo en una serie de partes o módulos más simples, facilitando así la programación y el depurado.
- **reutilización**: permite reutilizar una misma función en distintos programas.

Python dispone de una serie de *funciones integradas* (página 113) al lenguaje, y también permite crear funciones definidas por el usuario para ser usadas en su propios programas.

5.2.1 Sentencia def

La sentencia `def` es una definición de función usada para crear objetos **funciones definidas por el usuario**.

Una definición de función es una sentencia ejecutable. Su ejecución enlaza el nombre de la función en el *namespace* local actual a un objecto función (un envoltorio alrededor del código ejecutable para la función). Este objeto función contiene una referencia al *namespace* local global como el *namespace* global para ser usado cuando la función es llamada.

La definición de función no ejecuta el cuerpo de la función; esto es ejecutado solamente cuando la función es llamada.

La sintaxis para una definición de función en Python es:

```
def NOMBRE(LISTA_DE_PARAMETROS):
    """DOCSTRING_DE_FUNCION"""
    SENTENCIAS
    RETURN [EXPRESION]
```

A continuación se detallan el significado de pseudo código fuente anterior:

- `NOMBRE`, es el nombre de la función.

- LISTA_DE_PARAMETROS, es la lista de parámetros que puede recibir una función.
- DOCSTRING_DE_FUNCION, es la cadena de caracteres usada para *documentar* (página 50) la función.
- SENTENCIAS, es el bloque de sentencias en código fuente Python que realizar cierta operación dada.
- RETURN, es la *sentencia return* (página 104) en código Python.
- EXPRESION, es la expresión o variable que devuelve la sentencia `return`.

Un ejemplo simple de función esta seguidamente:

```
>>> def hola(arg):
...     """El docstring de la función"""
...     print "Hola", arg, "!"
...
>>> hola("Plone")
Hola Plone !
```

Advertencia: Los bloques de `function` deben estar indentado como otros bloques estructuras de control.

La palabra reservada `def` se usa para definir funciones. Debe seguirle el nombre de la función en el ejemplo anterior `hola()` y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar indentado.

La primer sentencia del cuerpo de la función puede ser opcionalmente una cadenas de caracteres literal; esta es la cadenas de caracteres de documentación de la función, o `docstrings`. (Puedes encontrar más acerca de `docstrings` en la sección *Cadenas de texto de documentación* (página 50)).

Hay herramientas que usan las `docstrings` para producir automáticamente documentación en línea o imprimible, o para permitirle al usuario que navegue el código en forma interactiva; es una buena práctica incluir `docstrings` en el código que uno escribe, por lo que se debe hacer un hábito de esto.

La ejecución de la función `hola()` muestra la impresión de un mensaje **Hola Plone !** que se imprime por consola. Devolver el objeto por los valores de retorno opcionales.

La ejecución de una función introduce una nueva tabla de símbolos usada para las variables locales de la función. Más precisamente, todas las asignaciones de variables en la función almacenan el valor en la tabla de símbolos local; así mismo la referencia a variables primero mira la tabla de símbolos local, luego en la tabla de símbolos local de las funciones externas, luego la tabla de símbolos global, y finalmente la tabla de nombres predefinidos. Así, no se les puede asignar directamente un valor a las variables globales dentro de una función (a menos se las nombre en la sentencia *global* (página 31)), aunque si pueden ser referenciadas.

Los parámetros reales (argumentos) de una función se introducen en la tabla de símbolos local de la función llamada cuando esta es ejecutada; así, los argumentos son pasados por valor (dónde el valor es siempre una referencia a un objeto, no el valor del objeto). Cuando una función llama a otra función, una nueva tabla de símbolos local es creada para esa llamada.

La definición de una función introduce el nombre de la función en la tabla de símbolos actual. El valor del nombre de la función tiene un tipo que es reconocido por el interprete como una función definida por el usuario. Este valor puede ser asignado a otro nombre que luego puede ser usado como una función. Esto sirve como un mecanismo general para renombrar.

5.2.2 Argumentos y parámetros

Al definir una función los valores los cuales se reciben se denominan parámetros, pero durante la llamada los valores que se envían se denominan argumentos.

Por posición

Cuando enviá argumentos a una función, estos se reciben por orden en los parámetros definidos. Se dice por tanto que son argumentos por posición:

```
>>> def resta(a, b):
...     return a - b
...
>>> resta(30, 10)
20
```

En el ejemplo anterior el argumento *30* es la posición *0* por consiguiente es el parámetro de la función *a*, seguidamente el argumento *10* es la posición *1* por consiguiente es el parámetro de la función *b*.

Por nombre

Sin embargo es posible evadir el orden de los parámetros si indica durante la llamada que valor tiene cada parámetro a partir de su nombre:

```
>>> def resta(a, b):
...     return a - b
...
>>> (b=30, a=10)
-20
```

Llamada sin argumentos

Al momento de llamar una función la cual tiene definidos unos parámetros, si no pasa los argumentos correctamente provocará una excepción *TypeError* (página 193):

```
>>> resta()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: resta() takes exactly 2 arguments (0 given)
```

Parámetros por defecto

Para solucionar la excepción *TypeError* (página 193) ejecutada al momento de la llamada a una función sin argumentos, entonces usted puede asignar unos valores por defecto nulos a los parámetros, de esa forma puede hacer una comprobación antes de ejecutar el código de la función:

```
>>> def resta(a=None, b=None):
...     if a == None or b == None:
...         print "Error, debes enviar dos números a la función"
...         return
...     return a - b
...
>>> resta(30, 10)
20
>>> resta()
Error, debes enviar dos números a la función
```

Como puede ver el el código anterior, se indica el final de la función luego de la *sentencia print* (página 153), usando la *sentencia return* (página 104) aunque no devuelva nada.

5.2.3 Argumentos indeterminados

En alguna ocasión usted no sabe previamente cuantos elementos necesita enviar a una función. En estos casos puede utilizar los parámetros indeterminados por posición y por nombre.

Por posición

Usted debe crear una lista dinámica de argumentos, es decir, un tipo *tupla* (página 63), definiendo el parámetro con un asterisco, para recibir los parámetros indeterminados por posición:

```
>>> def indeterminados_posicion(*args):
...     for arg in args:
...         print arg
...
>>> indeterminados_posicion(5,"Hola Plone",[1,2,3,4,5])
5
Hola Plone
[1, 2, 3, 4, 5]
```

Por nombre

Para recibir un número indeterminado de parámetros por nombre (clave-valor o en inglés *keyword args*), usted debe crear un diccionario dinámico de argumentos definiendo el parámetro con dos asteriscos:

```
>>> def indeterminados_nombre(**kwargs):
...     print kwargs
...
>>> indeterminados_nombre(n=5, c="Hola Plone", l=[1,2,3,4,5])
{'c': 'Hola Plone', 'l': [1, 2, 3, 4, 5], 'n': 5}
```

Al recibirse como un diccionario, puede iterarlo y mostrar la clave y valor de cada argumento:

```
>>> def indeterminados_nombre(**kwargs):
...     for kwarg in kwargs:
...         print kwarg, "=>", kwargs[kwarg]
...
>>> indeterminados_nombre(n=5, c="Hola Plone", l=[1,2,3,4,5])
c => Hola Plone
l => [1, 2, 3, 4, 5]
n => 5
```

Por posición y nombre

Si requiere aceptar ambos tipos de parámetros simultáneamente en una función, entonces debe crear ambas colecciones dinámicas. Primero los argumentos indeterminados por valor y luego los cuales son por clave y valor:

```
>>> def super_funcion(*args,**kwargs):
...     total = 0
...     for arg in args:
...         total += arg
...     print "sumatorio => ", total
...     for kwarg in kwargs:
...         print kwarg, "=>", kwargs[kwarg]
...
>>> super_funcion(50, -1, 1.56, 10, 20, 300, cms="Plone", edad=38)
sumatorio =>  380.56
edad => 38
cms => Plone
```

Los nombres `args` y `kwargs` no son obligatorios, pero se suelen utilizar por convención.

Muchos frameworks y librerías los utilizan por lo que es una buena practica llamarlos así.

5.2.4 Sentencia pass

Es una operación nula — cuando es ejecutada, nada sucede. Eso es útil como un contenedor cuando una sentencia es requerida sintácticamente, pero no necesita código que ser ejecutado, por ejemplo:

```
>>> # una función que no hace nada (aun)
... def consultar_nombre_genero(letra_genero): pass
...
>>> type(consultar_nombre_genero)
<type 'function'>
>>> consultar_nombre_genero("M")
>>>
>>> # una clase sin ningún método (aun)
... class Persona: pass
...
>>> macagua = Persona
>>> type(macagua)
<type 'classobj'>
```

5.2.5 Sentencia return

Las funciones pueden comunicarse con el exterior de las mismas, al proceso principal del programa usando la sentencia `return`. El proceso de comunicación con el exterior se hace *devolviendo valores*. A continuación, un ejemplo de función usando `return`:

```
def devuelve_fibonacci(n):
    '''devuelve la sucesión Fibonacci hasta n'''
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a + b
    return resultado
```

Esta función se llama de la siguiente forma:

```
>>> devuelve_fibonacci(10)
[1, 1, 2, 3, 5, 8]
```

Nota: Por defecto, las funciones retorna el valor *None* (página 210).

Retorno múltiple

Una característica interesante, es la posibilidad de devolver valores múltiples separados por comas:

```
>>> def prueba():
...     return "Plone CMS", 20, [1,2,3]
...
>>> prueba()
('Plone CMS', 20, [1, 2, 3])
```

En el código anterior los valores múltiples se tratan en conjunto como una *tupla* (página 63) inmutable y se pueden reasignar a distintas variables:

```
>>> def prueba():
...     return "Plone CMS", 20, [1,2,3]
...
>>> prueba()
('Plone CMS', 20, [1, 2, 3])
>>> cadena, numero, lista = prueba()
>>> print cadena, type(cadena)
Plone CMS <type 'str'>
>>> print numero, type(numero)
20 <type 'int'>
>>> print lista, type(lista)
[1, 2, 3] <type 'list'>
```

En el código anterior puede observa como se asignar a distintas variables en base a los valores de la *tupla* (página 63) inmutable.

5.2.6 Ejemplos de funciones

A continuación, se presentan algunos ejemplos de su uso:

Definición de funciones

A continuación, se presenta un ejemplo del uso de definir funciones:

```
def iva():
    '''función básica para el calculo del IVA'''
    iva = 12
    costo = int(input('¿Cual es el monto a calcular?: '))
    calculo = costo * iva / 100
    print ("El calculo de IVA es: " + str(calculo) + "\n")
```

Invocar funciones

A continuación, se presenta un ejemplo del uso de llamar funciones:

```
>>> iva()
¿Cual es el monto a calcular?: 300
36
```

Funciones con argumentos múltiple

A continuación, se presenta un ejemplo del uso de funciones con argumentos múltiple:

```
def suma(numero1, numero2):
    '''función la cual suma dos números'''
    print (numero1 + numero2)
    print ("\n")
```

Y se llama de la siguiente forma:

```
>>> suma(23, 74)
97
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `funciones.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python funciones.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 264) del entrenamiento para ampliar su conocimiento en esta temática.

5.3 Funciones avanzadas

En Python hay varias funciones avanzadas que se describen a continuación:

5.3.1 Funciones de predicado

Las funciones de predicado no es más que una función la cual dice si algo es `True` o `False`, es decir, es una función que devuelve un tipo de datos *booleano* (página 44).

Por hacer: TODO terminar de escribir la sección Funciones de predicado.

5.3.2 Objetos de función

Por hacer: TODO escribir la sección Objetos de función.

5.3.3 Funciones anónimas

Una función anónima, como su nombre indica es una función sin nombre. Es decir, es posible ejecutar una función sin referenciar un nombre, en Python puede ejecutar una función sin definirla con `def`.

De hecho son similares pero con una diferencia fundamental, **el contenido de una función anónima debe ser una única expresión en lugar de un bloque de acciones**.

Las funciones anónimas se implementan en Python con las funciones o expresiones *lambda* (página 106), esta es unas de las funcionalidades más potentes de Python, pero a la vez es la más confusas para los principiantes.

Más allá del sentido de función que usted tiene hasta el momento, con su nombre y sus acciones internas, una función en su sentido más trivial significa realizar algo sobre algo. Por tanto se podría decir que, mientras las funciones anónimas `lambda` sirven para realizar funciones simples, las funciones definidas con `def` sirven para manejar tareas más extensas.

5.3.4 Expresión lambda

Si deconstruye una función sencilla, puede llegar a una función `lambda`. Por ejemplo la siguiente función es para doblar un valor de un número:

```
>>> def doblar(numero):
...     resultado = numero*2
...     return resultado

>>> doblar(2)
4
>>> type(doblar)
<type 'function'>
```

Si el código fuente anterior se simplifica se verá, de la siguiente forma:

```
>>> def doblar(numero):
...     return numero*2

>>> doblar(2)
4
>>> type(doblar)
<type 'function'>
```

Usted puede todavía simplificar más, escribirlo todo en una sola línea, de la siguiente forma:

```
>>> def doblar(numero): return numero*2

>>> lambda numero: numero*2
<function <lambda> at 0x7f1023944e60>
>>> doblar(2)
4
>>> type(doblar)
<type 'function'>
```

Esta notación simple es la que una función `lambda` intenta replicar, observe, a continuación se va a convertir la función en una función anónima:

```
>>> lambda numero: numero*2
<function <lambda> at 0x7f1023944e60>
```

En este ejemplo tiene una función anónima con una entrada que recibe `numero`, y una salida que devuelve `numero * 2`.

Lo único que necesita hacer para utilizarla es guardarla en una variable y utilizarla tal como haría con una función normal:

```
>>> doblar = lambda numero: numero*2
>>> doblar(2)
4
>>> type(doblar)
<type 'function'>
```

Con la flexibilidad de Python usted puede implementar infinitas funciones simples. Usted puede encontrar más ejemplos de funciones anónimas usando `lambda` en la sección *ejemplos de funciones avanzadas* (página 107).

Usted puede explotar al máximo la función lambda utilizándola en conjunto con otras funciones como *filter()* (página 109) y *map()* (página 110).

5.3.5 Ejemplos de funciones avanzadas

A continuación, se presentan algunos ejemplos de su uso:

Función lambda - operaciones aritméticas

A continuación, se presenta un ejemplo para comprobar si un número es impar:

```
>>> impar = lambda numero: numero%2 != 0
>>> impar(5)
True
```

Función lambda - operaciones de cadena

A continuación, se presenta un ejemplo para darle la vuelta a una cadena rebanándola en sentido inverso:

```
>>> revertir = lambda cadena: cadena[::-1]
>>> revertir("Plone")
'enolP'
>>> revertir("enolP")
'Plone'
```

Función lambda - varios parámetros

A continuación, se presenta un ejemplo para varios parámetros, por ejemplo para sumar dos números:

```
>>> sumar = lambda x,y: x+y
>>> sumar(5,2)
7
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 264) del entrenamiento para ampliar su conocimiento en esta temática.

5.4 Funciones recursivas

Las funciones recursivas son funciones que se llaman a sí mismas durante su propia ejecución. Ellas funcionan de forma similar a las iteraciones, pero debe encargarse de planificar el momento en que dejan de llamarse a sí mismas o tendrá una función recursiva infinita.

Estas funciones se estilan utilizar para dividir una tarea en sub-tareas más simples de forma que sea más fácil abordar el problema y solucionarlo.

5.4.1 Función recursiva sin retorno

Un ejemplo de una función recursiva sin retorno, es el ejemplo de cuenta regresiva hasta cero a partir de un número:

```
>>> def cuenta_regresiva(numero):
...     numero -= 1
...     if numero > 0:
...         print numero
...         cuenta_regresiva(numero)
...     else:
...         print "Boooooooom!"
...     print "Fin de la función", numero
...
>>> cuenta_regresiva(5)
4
3
2
1
Boooooooom!
Fin de la función 0
Fin de la función 1
Fin de la función 2
Fin de la función 3
Fin de la función 4
```

5.4.2 Función recursiva con retorno

Un ejemplo de una función recursiva con retorno, es el ejemplo del calculo del factorial de un número corresponde al producto de todos los números desde 1 hasta el propio número. Es el ejemplo con retorno más utilizado para

mostrar la utilidad de este tipo de funciones:

```
>>> def factorial(numero):
...     print "Valor inicial ->",numero
...     if numero > 1:
...         numero = numero * factorial(numero -1)
...     print "valor final ->",numero
...     return numero
...
>>> print factorial(5)
Valor inicial -> 5
Valor inicial -> 4
Valor inicial -> 3
Valor inicial -> 2
Valor inicial -> 1
valor final -> 1
valor final -> 2
valor final -> 6
valor final -> 24
valor final -> 120
120
```

Por hacer: TODO terminar de escribir la sección Funciones recursivas.

Ver también:

Consulte la sección de *lecturas suplementarias* (página 264) del entrenamiento para ampliar su conocimiento en esta temática.

5.5 Funciones de orden superior

Las funciones de Python pueden tomar funciones como parámetros y devolver funciones como resultado. Una función que hace ambas cosas o alguna de ellas se llama función de orden superior.

5.5.1 filter()

La función `filter()` es una función la cual toma un *predicado* (página 106) y una lista y devuelve una lista con los elementos que satisfacen el predicado. Tal como su nombre indica `filter()` significa filtrar, ya que a partir de una lista o iterador y una función condicional, es capaz de devolver una nueva colección con los elementos filtrados que cumplan la condición.

Todo esto podría haberse logrado también usando *listas por comprensión* (página 231) que usaran *predicados*. No hay ninguna regla que diga cuando usar la función *map()* (página 110) o la función `filter()` en lugar de las *listas por comprensión* (página 231), simplemente debe decidir que es más legible dependiendo del contexto.

Por ejemplo, suponga que tiene una lista varios números y requiere filtrarla, quedando únicamente con los números múltiples de 5, eso seria así:

```
>>> # Primero declaramos una función condicional
def multiple(numero):
# Comprobamos si un numero es múltiple de cinco
if numero % 5 == 0:
    # Sólo devolvemos True si lo es
    return True

>>> numeros = [2, 5, 10, 23, 50, 33]
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> filter(multiple, numeros)
[5, 10, 50]
```

Si ejecuta el filtro obtiene una lista los números múltiples de 5. Por tanto cuando utiliza la función `filter()` tiene que enviar una función condicional, para esto, puede utilizar una función anónima `lambda`, como se muestra a continuación:

```
>>> numeros = [2, 5, 10, 23, 50, 33]
>>> filter(lambda numero: numero%5 == 0, numeros)
[5, 10, 50]
```

Así, en una sola línea ha definido y ejecutado el filtro utilizando una función condicional anónima y devolviendo una lista de números.

Filtrando objetos

Sin embargo, más allá de filtrar listas con valores simples, el verdadero potencial de la función `filter()` sale a relucir cuando usted necesita filtrar varios objetos de una lista.

Por ejemplo, dada una lista con varias personas, a usted le gustaría filtrar únicamente las cuales son menores de edad:

```
>>> class Persona:
...
...     def __init__(self, nombre, edad):
...         self.nombre = nombre
...         self.edad = edad
...
...     def __str__(self):
...         return "{} de {} años".format(self.nombre, self.edad)
...
>>> personas = [
...     Persona("Leonardo", 38),
...     Persona("Ana", 33),
...     Persona("Sabrina", 12),
...     Persona("Enrique", 3)
... ]
>>> menores = filter(lambda persona: persona.edad < 18, personas)
>>> for menor in menores:
print menor
Sabrina de 12 años
Enrique de 3 años
```

Este es un ejemplo sencillo, con el cual usted puede realizar filtrados con objetos, de forma amigable.

5.5.2 map()

La función `map()` toma una función y una lista y aplica esa función a cada elemento de esa lista, produciendo una nueva lista. Va a ver su definición de tipo y como se define.

Esta función trabaja de una forma muy similar a *filter()* (página 109), con la diferencia que en lugar de aplicar una condición a un elemento de una lista o secuencia, aplica una función sobre todos los elementos y como resultado se devuelve un lista de números doblado su valor:

```
>>> def doblar(numero):
return numero*2

>>> numeros = [2, 5, 10, 23, 50, 33]
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> map(doblar, numeros)
[4, 10, 20, 46, 100, 66]
```

Usted puede simplificar el código anterior usando una función `lambda` para substituir la llamada de una función definida, como se muestra a continuación:

```
>>> map(lambda x: x*2, numeros)
[4, 10, 20, 46, 100, 66]
```

La función `map()` se utiliza mucho junto a expresiones `lambda` ya que permite evitar escribir *bucles for* (página 90).

Además se puede utilizar sobre más de un objeto iterable con la condición que tengan la misma longitud. Por ejemplo, si requiere multiplicar los números de dos listas:

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [6, 7, 8, 9, 10]
>>> map(lambda x,y : x*y, a,b)
[6, 14, 24, 36, 50]
```

E incluso usted puede extender la funcionalidad a tres listas o más:

```
>>> a = [1, 2, 3, 4, 5]
>>> b = [6, 7, 8, 9, 10]
>>> c = [11, 12, 13, 14, 15]
>>> map(lambda x,y,z : x*y*z, a,b,c)
[66, 168, 312, 504, 750]
```

Mapeando objetos

Evidentemente, siempre que la función `map()` la utilice correctamente podrá mapear una serie de objetos sin ningún problema:

```
>>> class Persona:
...
...     def __init__(self, nombre, edad):
...         self.nombre = nombre
...         self.edad = edad
...
...     def __str__(self):
...         return "{} de {} años".format(self.nombre, self.edad)
...
>>> personas = [
...     Persona("Leonardo", 38),
...     Persona("Ana", 33),
...     Persona("Sabrina", 12),
...     Persona("Enrique", 3)
... ]
>>> def incrementar(p):
...     p.edad += 1
...     return p
...
>>> personas = map(incrementar, personas)
>>> for persona in personas:
...     print persona
...
Leonardo de 39 años
Ana de 34 años
Sabrina de 13 años
Enrique de 4 años
```

Claro que en este caso tiene que utilizar una función definida porque no necesitamos actuar sobre la instancia, a no ser que usted se tome la molestia de rehacer todo el objeto:

```
>>> class Persona:
...
...     def __init__(self, nombre, edad):
...         self.nombre = nombre
...         self.edad = edad
...
...     def __str__(self):
...         return "{} de {} años".format(self.nombre, self.edad)
...
>>> personas = [
...     Persona("Leonardo", 38),
...     Persona("Ana", 33),
...     Persona("Sabrina", 12),
...     Persona("Enrique", 3)
... ]
>>> def incrementar(p):
...     p.edad += 1
...     return p
...
>>> personas = map(lambda p: Persona(p.nombre, p.edad+1), personas)
>>> for persona in personas:
...     print persona
...
Leonardo de 39 años
Ana de 34 años
Sabrina de 13 años
Enrique de 4 años
```

5.5.3 lambda

La expresión `lambda`, es una función anónima que suelen ser usadas cuando necesita una función una sola vez. Normalmente usted crea funciones `lambda` con el único propósito de pasarlas a funciones de orden superior.

En muchos lenguajes, el uso de `lambdas` sobre funciones definidas causa problemas de rendimiento. No es el caso en Python.

```
>>> import os
>>> archivos = os.listdir(os.__file__.replace("/os.pyc", "/"))
>>> print filter(lambda x: x.startswith('os.'), archivos)
['os.pyc', 'os.py']
```

En el ejemplo anterior se usa el método `os.__file__` para obtener la ruta donde esta instalada el módulo `os` en su sistema, ejecutando la siguiente sentencia:

```
>>> os.__file__
'/usr/lib/python2.7/os.pyc'
```

Si con el método `os.__file__` obtiene la ruta del módulo `os` con el método `replace("/os.pyc", "/")` busca la cadena de carácter «/os.pyc» y la remplaza por la cadena de carácter «/»

```
>>> os.__file__.replace("/os.pyc", "/")
'/usr/lib/python2.7/'
```

Luego se define la variable `archivos` generando una lista de archivos usando la función `os.listdir()`, pasando el parámetro obtenido de la ruta donde se instalo el módulo `os` ejecutando en el comando previo, con la siguiente sentencia:

```
>>> archivos = os.listdir("/usr/lib/python2.7/")
```

De esta forma se define en la variable `archivos` un *tipo lista* (página 58) con un tamaño de *433*, como se puede comprobar a continuación:

```
>>> type(archivos)
<type 'list'>
>>> len(archivos)
443
```

Opcionalmente puede comprobar si la cadena de caracteres **os.pyc** se encuentras una de las posiciones de la lista `archivos`, ejecutando la siguiente sentencia:

```
>>> "os.pyc" in archivos
True
```

Ya al comprobar que existe la cadena de caracteres «**os.pyc**» se usa una función `lambda` como parámetro de la función *filter()* (página 109) para filtrar todos los archivos del directorio *«/usr/lib/python2.7/»* por medio del función `os.listdir()` que inicien con la cadena de caracteres «**os.**» usando la función *startswith()* (página 135).

```
>>> print filter(lambda x: x.startswith('os.'), os.listdir('/usr/lib/python2.7/'))
['os.pyc', 'os.py']
```

Así de esta forma se comprueba que existe el archivo compilado «**os.pyc**» de Python junto con el mismo módulo Python «**os.py**».

Truco: Más detalle consulte la referencia de las expresiones *lambda* (página 106).

Ver también:

Consulte la sección de *lecturas suplementarias* (página 264) del entrenamiento para ampliar su conocimiento en esta temática.

5.6 Funciones integradas

El interprete Python tiene un número de funciones integradas (built-in) dentro del módulo `__builtins__`, las cuales están siempre disponibles. Estas funciones están listadas en orden alfabéticos a continuación:

5.6.1 Funciones generales

Las funciones de uso general se describen a continuación:

apply()

La función `apply()` devuelve el resultado de una función o objeto clase llamado con argumentos soportados.

```
>>> def demo(valor1, valor2, valor3=None):
...     return valor1, valor2, valor3
...
>>> apply(demo, (1, 2), {'valor3': 3})
(1, 2, 3)
```

callable()

La función `callable()` le indica si un objecto puede ser llamado.

```
>>> callable([1,2,3])
False
>>> callable(callable)
True
>>> callable(False)
False
>>> callable(list)
True
```

Una función se puede llamar, una lista no se puede llamar. Incluso la función integrada `callable()` se puede llamar.

compile()

La función `compile()` devuelve un código objeto Python. Usted usa la función integrada Python para convertir de la cadena de caracteres de código al código objeto.

```
>>>
>>> exec(compile('a=5\nb=7\nprint a+b','','exec'))
12
```

Aquí, `exec` es el modo. El parámetro anterior que eso es el nombre del archivo para la forma del archivo el cual el código es leído. Finalmente, es ejecutado usando la función `exec()`.

credits()

Imprime el texto de la lista de contribuidores.

```
>>> credits()
    Thanks to CWI, CNRI, BeOpen.com, Zope Corporation and a cast of thousands
    for supporting Python development.  See www.python.org for more information.
```

copyright()

Imprime el texto de la nota de copyright.

```
>>> copyright()
Copyright (c) 2001-2016 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

dir()

Si es llamado sin argumentos, devuelve los nombres en el ámbito actual.

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
```

De lo contrario, devuelve una lista alfabética de nombres que comprende (alguno(s) de) los atributos de un objeto dato, y de los atributos legibles desde este.

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException', 'BufferError', 'BytesWarning',
'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
'None', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'ReferenceError',
'RuntimeError', 'RuntimeWarning', 'StandardError',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError',
'_', '__debug__', '__doc__', '__import__', '__name__',
'__package__', 'abs', 'all', 'any', 'apply', 'basestring',
'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter',
'float', 'format', 'frozenset', 'getattr', 'globals',
'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'intern', 'isinstance', 'issubclass', 'iter', 'len',
'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property', 'quit', 'range', 'raw_input', 'reduce', 'reload',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr',
'unicode', 'vars', 'xrange', 'zip']
```

Si el objeto soporta un método llamado `__dir__`, ese será usado; de lo contrario se usa la lógica `dir()` predeterminada y devuelve:

- para un objeto módulo: los atributos del módulo.

```
>>> import os
>>> type(os)
<type 'module'>
>>> dir(os)
['EX_CANTCREAT', 'EX_CONFIG', 'EX_DATAERR', 'EX_IOERR',
'EX_NOHOST', 'EX_NOINPUT', 'EX_NOPERM', 'EX_NOUSER',
'EX_OK', 'EX_OSERR', 'EX_OSFILE', 'EX_PROTOCOL',
'EX_SOFTWARE', 'EX_TEMPFAIL', 'EX_UNAVAILABLE', 'EX_USAGE',
'F_OK', 'NGROUPS_MAX', 'O_APPEND', 'O_ASYNC', 'O_CREAT',
'O_DIRECT', 'O_DIRECTORY', 'O_DSYNC', 'O_EXCL', 'O_LARGEFILE',
'O_NDELAY', 'O_NOATIME', 'O_NOCTTY', 'O_NOFOLLOW', 'O_NONBLOCK',
'O_RDONLY', 'O_RDWR', 'O_RSYNC', 'O_SYNC', 'O_TRUNC', 'O_WRONLY',
'P_NOWAIT', 'P_NOWAITO', 'P_WAIT', 'R_OK', 'SEEK_CUR', 'SEEK_END',
'SEEK_SET', 'ST_APPEND', 'ST_MANDLOCK', 'ST_NOATIME', 'ST_NODEV',
'ST_NODIRATIME', 'ST_NOEXEC', 'ST_NOSUID', 'ST_RDONLY',
'ST_RELATIME', 'ST_SYNCHRONOUS', 'ST_WRITE', 'TMP_MAX', 'UserDict',
'WCONTINUED', 'WCOREDUMP', 'WEXITSTATUS', 'WIFCONTINUED', 'WIFEXITED',
'WIFSIGNALED', 'WIFSTOPPED', 'WNOHANG', 'WSTOPSIG', 'WTERMSIG',
'WUNTRACED', 'W_OK', 'X_OK', '_Environ', '__all__', '__builtins__',
```

(continué en la próxima página)

(proviene de la página anterior)

```
'__doc__', '__file__', '__name__',
...
...
... ]
>>> print os.__doc__
OS routines for NT or Posix depending on what system we're on.

This exports:
  - all functions from posix, nt, os2, or ce, e.g. unlink, stat, etc.
  - os.path is one of the modules posixpath, or ntpath
  - os.name is 'posix', 'nt', 'os2', 'ce' or 'riscos'
  - os.curdir is a string representing the current directory ('.' or ':')
  - os.pardir is a string representing the parent directory ('..' or '::')
  - os.sep is the (or a most common) pathname separator ('/' or ':' or '\\')
...
...
...
```

- para un objeto clase: sus atributos, y recursivamente los atributos de sus clases bases.

```
>>> class Persona(object):
...     """Clase que representa una Persona"""
...     def __init__(self, cedula, nombre, apellido, sexo):
...         """ Constructor de clase Persona """
...         self.cedula = cedula
...         self.nombre = nombre
...         self.apellido = apellido
...         self.sexo = sexo
...     def __str__(self):
...         """Devuelve una cadena representativa al Persona"""
...         return "%s: %s %s, %s." % (
...             str(self.cedula), self.nombre,
...             self.apellido, self.sexo
...         )
...     def hablar(self, mensaje):
...         """Mostrar mensaje de saludo de Persona"""
...         print mensaje
...
>>> type(Persona)
<type 'type'>
>>> vars()
{'Persona': <class '__main__.Persona'>,
'__builtins__': <module '__builtin__' (built-in)>,
'__package__': None, '__name__': '__main__',
'os': <module 'os' from '/usr/lib/python2.7/os.pyc'>,
'__doc__': None}
>>> dir(Persona)
['__class__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'hablar']
>>> Persona.__dict__
dict_proxy({'__module__': '__main__',
'__str__': <function __str__ at 0x7fab8aaad758>,
'__dict__': <attribute '__dict__' of 'Persona' objects>,
'hablar': <function hablar at 0x7fab8aaad7d0>,
'__weakref__': <attribute '__weakref__' of 'Persona' objects>,
'__doc__': ' Clase que representa una persona. ',
'__init__': <function __init__ at 0x7fab8aaad6e0>})
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> Persona.__doc__
' Clase que representa una persona. '
>>> Persona.__init__.__doc__
' Constructor de clase Persona '
>>> Persona.hablar.__doc__
' Mostrar mensaje de saludo de Persona '
```

- para cualquier otro objecto: sus atributos, sus atributos de clases, y recursivamente los atributos de esas clases bases de las clases.

```
>>> type(int)
<type 'type'>
>>> dir(int)
['__abs__', '__add__', '__and__', '__class__', '__cmp__',
'__coerce__', '__delattr__', '__div__', '__divmod__',
'__doc__', '__float__', '__floordiv__', '__format__',
'__getattribute__', '__getnewargs__', '__hash__', '__hex__',
'__index__', '__init__', '__int__', '__invert__', '__long__',
'__lshift__', '__mod__', '__mul__', '__neg__', '__new__',
'__nonzero__', '__oct__', '__or__', '__pos__', '__pow__',
'__radd__', '__rand__', '__rdiv__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__',
'__rmod__', '__rmul__', '__ror__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'imag', 'numerator',
'real']
```

eval()

Evalúa una cadena como una expresión:

```
>>> eval('2 + 5')
7
```

Ademas si se han definido anteriormente variables las acepta como parámetros:

```
>>> numero = 10
>>> eval('numero * 10 - 5')
95
```

execfile()

La función `execfile()` lee y ejecuta un script Python desde un archivo. Los `globals` y `locals` son diccionarios, por defecto a los actuales `globals` y `locals`. Si solamente `globals` es dado, `locals` es por defecto a la misma.

```
>>> execfile('./holamundo.py')
Hola Mundo
```

globals()

La función `globals()` devuelve un diccionario conteniendo ámbito actual global de las variables.

```
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>,
'__package__': None, '__name__': '__main__', '__doc__': None}
```

La función `globals()` puede ser usada para devolver los nombres en el `namespaces` global dependiendo en la locación desde donde ella es llamada.

Si la función `globals()` es llamada desde una función, eso devolverá todos los nombres que pueden ser accesibles globalmente desde esa función.

El tipo de dato devuelto por función es un tipo diccionario. Por lo tanto, los nombres pueden ser extraídos usando la función integrada `keys()`.

help()

Invoca el menú de ayuda del intérprete de Python:

```
>>> help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics, type "modules",
"keywords", or "topics".  Each module also comes with a one-line summary
of what it does; to list the modules whose summaries contain a given word
such as "spam", type "modules spam".

help>
```

id()

La función `id()` devuelve la identidad de un objecto. Esto garantiza ser el único entre objetos simultáneamente existentes. (Sugerencia: es la dirección de memoria del objeto).

```
>>> lista = range(5)
>>> lista
[0, 1, 2, 3, 4]
>>> id(lista)
139703096777904
```

len()

Devuelve el número de elementos de un tipo de secuencia o colección.

```
>>> len("leonardo caballero")
18
```

license()

Imprime el texto de la licencia.

```
>>> license
Type license() to see the full license text
>>> license()
A. HISTORY OF THE SOFTWARE
==========================

Python was created in the early 1990s by Guido van Rossum at Stichting
Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands
as a successor of a language called ABC.  Guido remains Python's
principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for
National Research Initiatives (CNRI, see http://www.cnri.reston.va.us)
in Reston, Virginia where he released several versions of the
software.

In May 2000, Guido and the Python core development team moved to
BeOpen.com to form the BeOpen PythonLabs team.  In October of the same
year, the PythonLabs team moved to Digital Creations (now Zope
Corporation, see http://www.zope.com).  In 2001, the Python Software
Foundation (PSF, see https://www.python.org/psf/) was formed, a
non-profit organization created specifically to own Python-related
Intellectual Property.  Zope Corporation is a sponsoring member of
the PSF.

All Python releases are Open Source (see http://www.opensource.org for
Hit Return for more, or q (and Return) to quit:
```

locals()

La función `locals()` devuelve un diccionario conteniendo ámbito actual local de las variables.

```
>>> locals()
{'__builtins__': <module '__builtin__' (built-in)>,
'__package__': None, '__name__': '__main__', '__doc__': None}
```

La función `locals()` puede ser usadas para devolver los nombres en el `namespaces` local dependiendo en la locación desde donde ella es llamada.

Si la función `locals()` es llamada desde una función, eso devolverá todos los nombres que pueden ser accesibles localmente desde esa función.

El tipo de dato devuelto por la función es un tipo diccionario. Por lo tanto, los nombres pueden ser extraídos usando la función integrada `keys()`.

open()

La función `open()` es definida dentro del modulo integrado `io`, esta le permite *abrir un archivo* (página 154) usando el tipo objeto `file`, devuelve un objeto del tipo *file* (página 214) (ej. *archivo*), y se llama habitualmente con de dos a tres argumentos:

```
file(nombre[, modo[, buffering]]) -> objeto archivo
```

Los argumentos son:

- `nombre`, es una *cadena de caracteres* (página 46) que indica el *nombre de archivo* (incluso ruta relativa o absoluta).
- `modo`, es una cadena de unos pocos caracteres describiendo la forma en la que se usará el archivo, como se indica a continuación:

Modo	Notas
r	el archivo se abre en modo de solo lectura, no se puede escribir (argumento por defecto).
w	modo de solo escritura (si existe un archivo con el mismo nombre, se borra).
a	modo de agregado (append), los datos escritos se agregan al final del archivo.
r+	el archivo se abre para lectura y escritura al mismo tiempo.
b	el archivo se abre en modo binario, para almacenar cualquier cosa que no sea texto.
U	el archivo se abre con soporte a nueva linea universal, cualquier fin de linea ingresada sera como un \n en Python.

- buffering, si este argumento es dado, 0 significa sin búfer, 1 significa búfer de línea y los números más grandes especifican el tamaño del búfer.

Para crear y abrir un archivo, seria así:

```
>>> archivo = open('datos.txt', 'w')
>>> type(archivo)
<type 'file'>
```

El archivo será creado si no existe cuando es abierto para escribir o agregar data. Es archivo sera truncado cuando es abierto para escritura.

Agregue una "U" a modo para abrir el archivo para la entrada con soporte de nueva línea universal. Cualquier línea que termine en el archivo de entrada se verá como "n" en Python. Además, un archivo así abierto gana el atributo newlines; el valor para este atributo es uno de Ninguno (aún no se ha leído una nueva línea), \r, \n, \r\n o una tupla que contiene todos los tipos de nueva línea que se han visto.

Truco: Ver para futura información desde el *modo interactivo* (página 15) Python, lo siguiente:

```
>>> file.__doc__
```

---

range()

La función range() devuelve una lista conteniendo una progresión aritmética de enteros.

range(inicio, detener[, paso]) -> lista de enteros

```
>>> range(3,9)
[3, 4, 5, 6, 7, 8]
```

range(i, j) devuelve [i, i+1, i+2, ..., j-1]; inicia (!) por defecto en **0**.

Cuando el paso es definido como un tercer argumento, ese especifica el incremento (o decremento).

```
>>> range(3,9,2)
[3, 5, 7]
```

En el ejemplo anterior, la función range(3,9,2) devuelve **[3, 5, 7]**, es decir, el rango inicia en **3** y termina en **9** incrementando cada **2** números.

range(detener) -> lista de enteros

```
>>> range(4)
[0, 1, 2, 3]
```

En el ejemplo anterior, la función range(4) devuelve **[0, 1, 2, 3]**. ¡El punto final es omitido! Hay exactamente los indices validos para una lista de **4** elementos.

reload()

Cuando el modulo es importado dentro de un script, el código en la porción del nivel superior de un modulo es ejecutado solamente una vez.

Por lo tanto, si usted quiere volver a ejecutar la porción del nivel superior el código de un modulo, usted puede usar la función `reload()`. Esta función importa otra vez un modulo previamente importado. La sintaxis de la función `reload()` es la siguiente:

```
>>> reload(module_name)
```

Aquí, `module_name` es el nombre del modulo que usted quiere volver a cargar y no la *cadena de caracteres* (página 46) contendiente el nombre del modulo. Por ejemplo, para recargar el modulo `clases.py`, debe hacer lo siguiente:

```
>>> import clases
>>> reload(clases)
```

xrange()

El tipo `xrange` es un tipo secuencia inmutable utilizada normalmente en bucles. La ventaja de la función `xrange()` sobre la función `range()`, es que devuelve un objeto `xrange` el cual ocupa siempre la misma cantidad de memoria, independientemente del rango el cual represente.

```
>>> for item in range(5):
...     print item
...
0
1
2
3
4
>>> for item in xrange(5):
...     print item
...
0
1
2
3
4
>>>
```

Como la función `xrange()`, devuelve un objeto el cual genera los números en el rango a demanda. Para bucles, esto es un poco mas rápido que la función `range()` y más eficiente en la memoria.

```
>>> print xrange(5)
xrange(5)
>>> type(xrange(5))
<type 'xrange'>
>>> dir(xrange(5))
['__class__', '__delattr__', '__doc__', '__format__',
'__getattribute__', '__getitem__', '__hash__', '__init__',
'__iter__', '__len__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
```

La ventaja de la función `xrange()` es *excepto* en hardware impedido en cuestión de memoria (por ejemplo, MS-DOS) o cuando nunca se utilizan todos los elementos del rango (por ejemplo, porque se suele interrumpir la ejecución del bucle con la sentencia *break* (página 89)).

type()

La función `type()` devuelve el tipo del objeto que recibe como argumento.

```
>>> type(2)
<type 'int'>
>>> type(2.5)
<type 'float'>
>>> type(True)
<type 'bool'>
>>> type("Hola Mundo")
<type 'str'>
>>> type(int)
<type 'type'>
>>> type(str)
<type 'type'>
>>> type(None)
<type 'NoneType'>
>>> type(object)
<type 'type'>
>>> import os
>>> type(os)
<type 'module'>
>>> type(format)
<type 'builtin_function_or_method'>
```

Truco: La función `type()` devuelve el tipo del objeto, en base al modulo integrado `types`, el cual define los nombres para todos los símbolos tipo conocidos en el interprete estándar.

```
>>> import types
>>> help(types)

Help on module types:

NAME
    types - Define names for all type symbols known in the standard interpreter.

FILE
    /usr/lib/python2.7/types.py

MODULE DOCS
    https://docs.python.org/library/types

DESCRIPTION
    Types that are part of optional modules (e.g. array) are not listed.

CLASSES
    __builtin__.basestring(__builtin__.object)
        __builtin__.str
        __builtin__.unicode

>>>
```

vars()

La función `vars()` devuelve un diccionario conteniendo ámbito actual de las variables.

```
>>> vars()
{'__builtins__': <module '__builtin__' (built-in)>, '__package__':
None, '__name__': '__main__', '__doc__': None}
```

La función `vars()` sin argumentos, equivale a la función *locals()* (página 119). Si se llama con un argumento equivale a la sentencia `object.__dict__`.

5.6.2 Funciones de entrada y salida

Las funciones de tipos numéricos se describen a continuación:

input()

Equivalente a la función `eval(raw_input(prompt))`

Lee una *cadena de caracteres* (página 46) desde la entrada estándar.

```
>>> dato = input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: 2
2
<type 'int'>
>>> dato = input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: 23.4
23.4
<type 'float'>
>>> dato = input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: 23L
23L
<type 'long'>
```

En el caso que quiera ingresar una *cadena de caracteres* (página 46) desde la entrada estándar usando la función `input()`, debe colocar la cadena de caracteres entre comillas simples o dobles, como el siguiente ejemplo:

```
>>> dato = input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: leonardo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'leonardo' is not defined
>>> dato = input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: "leonardo"
'leonardo'
<type 'str'>
>>> dato = input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: leonardo caballero
  File "<string>", line 1
    leonardo caballero
                     ^
SyntaxError: unexpected EOF while parsing
>>> dato = input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: "leonardo caballero"
'leonardo caballero'
<type 'str'>
```

raw_input()

Lee una *cadena de caracteres* (página 46) desde la entrada estándar. La nueva línea final es despojada. Si el usuario indica un EOF (*Unix*: `Ctl-D`, *Windows*: `Ctl-Z+Return`), lanza una excepción *EOFError* (página 191). En

sistemas Unix, la librería **GNU readline** es usada si es habilitada. El `prompt` de la cadena de caracteres, si es dado, es impreso sin una nueva línea final antes de leer.

```
>>> dato = raw_input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: 2
'2'
<type 'str'>
>>> dato = raw_input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: 2.3
'2.3'
<type 'str'>
>>> dato = raw_input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: 23L
'23L'
<type 'str'>
>>> dato = raw_input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: leonardo
'leonardo'
<type 'str'>
>>> dato = raw_input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: "leonardo"
'"leonardo"'
<type 'str'>
>>> dato = raw_input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: leonardo caballero
'leonardo caballero'
<type 'str'>
>>> dato = raw_input("Por favor, ingresa un dato: "); dato; type(dato)
Por favor, ingresa un dato: "leonardo caballero"
'"leonardo caballero"'
<type 'str'>
```

5.6.3 Funciones numéricas

Las funciones de tipos numéricos se describen a continuación:

abs()

Devuelve el valor absoluto de un número (entero o de coma flotante).

```
>>> abs(3)
3
>>> abs(-3)
3
>>> abs(-2.5)
2.5
```

bin()

Devuelve una representación binaria de un *número entero* (página 41) o *entero long* (página 41), es decir, lo convierte de entero a binario.

```
>>> bin(10)
'0b1010'
```

cmp()

La función `cmp()` devuelve un valor negativo si `x<y`, un valor cero si `x==y`, un valor positivo si `x>y`:

```
>>> cmp(1,2)
-1
>>> cmp(2,2)
0
>>> cmp(2,1)
1
```

complex()

La función `complex()` devuelve un número complejo `complex`. Es un constructor, que crea un *entero complex* (página 43) a partir de un *entero* (página 41), *entero long* (página 41), *entero float* (página 42) (cadenas de caracteres formadas por números y hasta un punto), o una *cadena de caracteres* (página 46) que sean coherentes con un número entero.

```
>>> complex(23)
(23+0j)
>>> complex(23L)
(23+0j)
>>> complex(23.4)
(23.4+0j)
>>> complex("23")
(23+0j)
>>> complex("23.6")
(23.6+0j)
```

La función `complex()` sólo procesa correctamente cadenas que contengan exclusivamente números.Si la cadena contiene cualquier otro carácter, la función devuelve una excepción *ValueError* (página 193).

```
>>> complex("qwerty")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: complex() arg is a malformed string
```

divmod()

Debe recibir dos argumentos numéricos, y devuelve dos valores: resultado de la división entera, y el resto.

```
>>> divmod(22, 4)
(5, 2)
```

float()

La función `float()` devuelve un número coma flotante `float`. Es un constructor, que crea un *coma flotante* (página 42) a partir de un *entero* (página 41), *entero long* (página 41), *entero float* (página 42) (cadenas de caracteres formadas por números y hasta un punto) o una *cadena de caracteres* (página 46) que sean coherentes con un número entero.

```
>>> float(2)
2.0
>>> float(23L)
23.0
>>> float(2.5)
2.5
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> float("2")
2.0
>>> float("2.5")
2.5
```

hex()

Devuelve una representación hexadecimal de un *número entero* (página 41) o *entero long* (página 41), es decir, lo convierte de entero a hexadecimal.

```
>>> hex(10)
'0xa'
```

int()

La función `int()` devuelve un número entero. Es un constructor, que crea un *entero* (página 41) a partir de un *entero float* (página 42), *entero complex* (página 43) o una *cadena de caracteres* (página 46) que sean coherentes con un número entero.

```
>>> int(2.5)
2
```

También puede convertir una cadena de caracteres a un número entero.

```
>>> int("23")
23
```

La función `int()` sólo procesa correctamente cadenas que contengan exclusivamente números. Si la cadena contiene cualquier otro carácter, la función devuelve una excepción *ValueError* (página 193).

```
>>> int("2.5")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2.5'
>>>
>>> int("doscientos")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'doscientos'
```

long()

La función `long()` devuelve un número entero `long`. Es un constructor, que crea un *entero long* (página 41) a partir de un *entero* (página 41), *entero float* (página 42) o una *cadena de caracteres* (página 46) que sean coherentes con un número entero.

```
>>> long(23)
23L
>>> long(23.4)
23L
```

También puede convertir una cadena de caracteres a un número entero.

```
>>> long("23")
23
```

La función `long()` sólo procesa correctamente cadenas que contengan exclusivamente números. Si la cadena contiene cualquier otro carácter, la función devuelve una excepción *ValueError* (página 193).

```
>>> long("23.4")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for long() with base 10: '23.4'
>>>
>>> long("23,4")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for long() with base 10: '23,4'
```

max()

Si recibe más de un argumento, devuelve el mayor de ellos.

```
>>> max(23, 12, 145, 88)
145
>>> type(max(23, 12, 145, 88))
<type 'int'>
>>> max("a", "Z")
'a'
>>> type(max("a", "Z"))
<type 'str'>
```

Si recibe un solo argumento, devuelve el mayor de sus elementos. Debe ser un objeto iterable; puede ser una *cadena de caracteres* (página 46), o alguno de los otros tipos de secuencia o colección.

```
>>> max("Hola, Plone")
'o'
>>> type(max("Hola, Plone"))
<type 'str'>
```

min()

Tiene un comportamiento similar a `max()`, pero devuelve el mínimo.

```
>>> min(23, 12, 145, 88)
12
>>> type(min(23, 12, 145, 88))
<type 'int'>
>>> min("Hola, Plone")
' '
>>> type(min("Hola, Plone"))
<type 'str'>
```

pow()

La función `pow()` si recibe dos (02) argumentos, eleva el primero argumento a la potencia del segundo argumento.

```
>>> pow(2, 3)
8
>>> pow(10, 2)
100
>>> pow(10, -2)
0.01
```

Si recibe un tercer argumento opcional, éste funciona como módulo.

```
>>> pow(2, 3, 3)
2
```

reduce()

La función `reduce()` aplica una función de dos argumentos de forma acumulativa a los elementos de un tipo de secuencia, de izquierda a derecha, para reducir la secuencia a un solo valor. La sintaxis seria la siguiente:

```
>>> reduce(funcion, secuencia[, inicial]) -> valor
```

A continuación un ejemplo:

```
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
15
>>> ((((1+2)+3)+4)+5)
15
```

En el ejemplo anterior, calcula el siguiente calculo `(((((1+2)+3)+4)+5)`.

Si el argumento `inicial` está presente, se coloca antes de los elementos de la secuencia en el cálculo y sirve como valor predeterminado cuando la secuencia está vacía.

```
>>> reduce(lambda x, y: x + y, [1, 2, 3, 4, 5], 5 * 5)
40
```

En el ejemplo anterior, la función, usada es `lambda x, y: x + y`, la secuencia es la lista `[1, 2, 3, 4, 5]` y el argumento `inicial` es `5 * 5`

```
>>> reduce(lambda x, y: x + y, [0, 0, 0, 0, 0], 5 * 5)
25
```

En el ejemplo anterior, la función, usada es `lambda x, y: x + y`, la secuencia es la lista `[0, 0, 0, 0, 0]` y el argumento `inicial` es `5 * 5`

round()

La función `round()` redondea un número flotante a una precisión dada en dígitos decimal (por defecto 0 dígitos). Esto siempre devuelve un número flotante. La precisión tal vez sea negativa.

En el siguiente ejemplo redondeo de un número flotante a entero, mayor o igual a *.5* al alza:

```
>>> round(5.5)
6.0
```

En este otro ejemplo redondeo de un número flotante a entero, menor de *.5* a la baja:

```
>>> round(5.4)
5.0
```

sum()

La función `sum()` devuelve una lista ordenada de los elementos de la secuencia que recibe como argumento (lista o cadena). La secuencia original no es modificada.

```
>>> lista = [1, 2, 3, 4]
>>> sum(lista)
10
```

oct()

La función `oct()` convierte un número entero en una cadena en base octal, antecedida del prefijo *"0"*.

```
>>> oct(8)
'010'
>>> oct(123)
'0173'
```

5.6.4 Funciones de booleanos

Las funciones de tipos *booleanos* (página 44) se describen a continuación:

bool()

La función `bool()`, es un constructor, el cual crea un tipo de datos *booleanos* (página 44), devuelve un tipo booleano `True` cuando el argumento dado es `True`, de lo contrario `False`.

```
>>> bool()
False
>>> bool(True)
True
```

Convertir desde un tipo *entero* (página 40) a tipo *booleano*:

```
>>> bool(0)
False
>>> bool(1)
True
```

Convertir desde un tipo *entero float* (página 42) de forma recursiva usando la función *int()* (página 126) a tipo *booleano*:

```
>>> bool(int(0.1))
False
>>> bool(int(1.0))
True
```

Convertir desde un tipo *cadena de caracteres* (página 46) de forma recursiva usando la función *str()* (página 136) y la función *int()* (página 126) a tipo *booleano*:

```
>>> bool(int(str('0')))
False
>>> bool(int(str('1')))
True
```

5.6.5 Funciones de cadenas de caracteres

Las funciones de tipos *cadena de caracteres* (página 46) se describen a continuación:

capitalize()

La función `capitalize()` devuelve una *cadenas de caracteres* (página 46) con MAYÚSCULA la primera palabra.

```
>>> 'leonardo caballero'.capitalize()
'Leonardo caballero'
```

chr()

La función `chr()` recibe como argumento un entero, y devuelve una cadena con el carácter cuyo código *Unicode* corresponde a ese valor. El rango válido para el argumento es de 0 a 256.

```
>>> chr(64)
'@'
>>> chr(36)
'$'
>>> chr(94)
'^'
>>> chr(126)
'~'
```

endswith()

La función `endswith()` devuelve un valor booleano `True` o `False` si coincide que la cadena termine con el criterio enviado por parámetros en la función.

```
>>> 'leonardo caballero'.endswith("do")
False
>>> 'leonardo caballero'.endswith("ro")
True
```

expandtabs()

La función `expandtabs()` devuelve una copia de la *cadena de caracteres* (página 46) donde todos los caracteres `tab` (tabulación) son remplazados por uno o más espacios, depende en la actual columna y el tamaño del tab dado.

```
>>> 'Leonardo Caballero\tPython Developer\tleonardoc@plone.org'.expandtabs()
'Leonardo Caballero      Python Developer        leonardoc@plone.org'
```

Usted puede indicar el tamaño de la tecla `tab` vía parámetro de la función:

```
>>> 'Leonardo Caballero\tPython Developer\tleonardoc@plone.org'.expandtabs(4)
'Leonardo Caballero  Python Developer    leonardoc@plone.org'
>>> 'Leonardo Caballero\tPython Developer\tleonardoc@plone.org'.expandtabs(2)
'Leonardo Caballero  Python Developer  leonardoc@plone.org'
```

find()

La función `find()` devuelve un valor numérico `0` si encuentra el criterio de búsqueda o `-1` si no coincide el criterio de búsqueda enviado por parámetros en la función.

```
>>> 'leonardo caballero'.find("leo")
0
>>> 'leonardo caballero'.find("ana")
-1
```

format()

La función integrada `format()` devuelve una representación formateada de un valor dato controlado por el especificador de formato.

La función integrada `format()` es similar al *método format()* (página 53) disponible en el tipo de *cadena de caracteres* (página 46). Internamente, ambos llaman al método `__format__()` de un objecto.

Mientras, la función integrada `format()` es una implementación de bajo nivel para formatear un objeto usando `__format__()` internamente, el *método format()* (página 53) del tipo de cadena de caracteres es una implementación de alto nivel disponible para ejecutar operaciones de formateo complejas en múltiples objeto de *cadena de caracteres* (página 46).

La sintaxis de la función integrada `format()` es:

```
format(value[, format_spec])
```

La a función integrada `format()` toma dos parámetros:

- value - valor que necesita formatear.
- format_spec - La especificación en como el valor debe ser formateado.

A continuación, un ejemplo de un valor *número entero* (página 41), seria de la siguiente forma:

```
>>> print format(123,"d")
123
```

A continuación, un ejemplo de un valor *número float* (página 42), seria de la siguiente forma:

```
>>> print format(123.456789,"f")
123.456789
```

A continuación, un ejemplo de un valor binario, seria de la siguiente forma:

```
>>> print format(10,"b")
1010
```

A continuación, un ejemplo de un valor *número entero* (página 41) con formato especifico, seria de la siguiente forma:

```
>>> print format(1234,"*>+7,d")
*+1,234
```

En el ejemplo anterior cuando se formatea el *número entero* (página 41) *1234*, usted especifico el especificador de formato `*>+7,d`. Seguidamente, se describe cada opción a continuación:

- `*` Es la opción del carácter de relleno, el cual rellena los espacio vacío después del formato.
- `>` Es la opción de alineación a la derecha, el cual alinea la cadena de caracteres de salida a la derecha.
- `+` Es la opción de signo, el cual obliga al número a ser firmado (con un signo a su izquierda).
- `7` Es la opción ancho, el cual obliga el número que tome un mínimo de ancho de 7, otros espacios serán rellenado por el carácter de relleno.
- `,` Ese es el operador miles, el cual coloca un carácter coma entre todos los números miles.
- `d` Es la opción tipo que especifica que el número es un *número entero* (página 41).

A continuación, un ejemplo de un valor *número float* (página 42) con formato especifico, seria de la siguiente forma:

```
>>> print format(123.4567, "^-09.3f")
0123.4570
```

En el ejemplo anterior cuando se formatea el *número float* (página 42) *123.4567*, usted especifico el especificador de formato `^-09.3f`. Seguidamente, se describe cada opción a continuación:

- `^` Es la opción de alineación centrar, el cual alinea la cadena de caracteres de salida al centro del espacio restante.
- `-` Es la opción de signo el cual obliga solo a los números negativos a mostrar el signo.
- `0` Ese es el carácter, el cual es colocado en lugar de los espacios vacíos.
- `9` Es la opción de ancho, el cual establece el ancho mínimo del número en 9 (incluido el punto decimal, la coma y el signo de miles).
- `.3` Ese es el operador de precisión que define la precisión del número flotante dado a 3 lugares.
- `f` Es la opción tipo que especifica que el número es un *número float* (página 42).

A continuación, un ejemplo de usar la función `format()` sobre escribiendo el método especial `__format__()` de una *clase* (página 199), seria de la siguiente forma:

```
>>> class Persona:
...     def __format__(self, formato):
...         if(formato == 'edad'):
...             return '23'
...         return 'Formato nulo'
...
>>> print format(Persona(), "edad")
23
```

En el ejemplo anterior cuando se sobre escribe el método especial `__format__()` de la clase `Persona`. Ese ahora acepta el argumento del método llamado `edad` el cual devuelve *23*.

El método `format()` internamente ejecuta `Persona().__format__("edad")`, el cual devuelve el mensaje *23*. Si no hay formato especificado, el mensaje devuelto es *Formato nulo*.

index()

La función `index()` es como la función `find()` pero arroja una excepción *ValueError* (página 193) cuando la sub-cadena no es encontrada.

```
>>> 'leonardo caballero'.index("leo")
0
>>> 'leonardo caballero'.index("ana")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> 'leonardo caballero'.index(" ca")
8
```

intern()

La función `intern()` introduce la cadena en la tabla de cadenas internadas (si no está ya allí). Esto ingresa la cadena en la tabla (global) de cadenas internas cuyo propósito es acelerar las búsquedas en el tipo diccionario.

Al utilizar la función `intern()`, se asegura de que nunca cree dos objetos de cadena de caracteres que tengan el mismo valor: cuando solicita la creación de un segundo objeto de cadena de caracteres con el mismo valor que un objeto de cadena existente, recibe una referencia al objeto de cadena preexistente. De esta manera, estás ahorrando

memoria. Además, la comparación de objetos de cadena de caracteres ahora es muy eficiente porque se lleva a cabo comparando las direcciones de memoria de los dos objetos de cadena de caracteres en lugar de su contenido.

Esencialmente, la función `intern()` busca (o almacena si no está presente) la cadena de caracteres en una colección de cadenas de caracteres internadas, por lo que todas las instancias internadas compartirán la misma identidad. Cambia el costo único de buscar esta cadena de caracteres para realizar comparaciones más rápidas (la comparación puede devolver `True` después de solo verificar la identidad, en lugar de tener que comparar cada carácter), y reducir el uso de la memoria.

Sin embargo, Python internará automáticamente cadenas de caracteres que sean pequeñas o que parezcan identificadores, por lo que es posible que no obtengas ninguna mejora porque tus cadenas de caracteres ya están internadas entre bastidores.

A continuación uno ejemplo de comparación de cadena de caracteres con operadores de relacionales:

```
>>> cadena0, cadena1 = 'python', 'python'
>>> cadena0 == cadena1
True
>>> cadena0 is cadena1
True
>>> cadena0, cadena1 = 'python 2.7', 'python 2.7'
>>> cadena0 is cadena1
False
```

A continuación uno ejemplo de comparación de cadena de caracteres con el operador *is* (página 84):

```
>>>
>>> cadena0 = intern('plone cms')
>>> cadena1 = 'plone cms'
>>> cadena0 is cadena1
False
>>> cadena1 = intern('plone cms')
>>> cadena0 is cadena1
True
```

isalnum()

La función `isalnum()` devuelve un valor booleano `True` o `False` si coincide que la cadena contenga caracteres alfanuméricos.

```
>>> '23456987'.isalnum()
True
>>> 'V-23456987'.isalnum()
False
```

isalpha()

La función `isalpha()` devuelve un valor booleano `True` o `False` si coincide que la cadena contenga caracteres alfabéticos.

```
>>> 'leonardo'.isalpha()
True
>>> 'leonardo caballero'.isalpha()
False
```

isdigit()

La función `isdigit()` devuelve un valor booleano `True` o `False` si coincide que la cadena contenga caracteres dígitos.

```
>>> 'leonardo caballero'.isdigit()
False
>>> '23456987'.isdigit()
True
```

islower()

La función `islower()` devuelve un valor booleano `True` o `False` si coincide que la cadena contenga caracteres en MINÚSCULAS.

```
>>> 'leonardo caballero'.islower()
True
>>> 'leonardo CABALLERO'.islower()
False
```

istitle()

La función `istitle()` devuelve un valor booleano `True` o `False` si coincide que la *cadena de caracteres* (página 46) sean capitales en cada palabra.

```
>>> "leonardo caballero".title()
'Leonardo Caballero'
>>> "leonardo Caballero".istitle()
False
```

isspace()

La función `isspace()` devuelve un valor booleano `True` o `False` si no es vacía, y todos sus caracteres son espacios en blanco.

```
>>> " ".isspace()
True
>>> "  ".isspace()
True
>>> "a ".isspace()
False
>>> " A ".isspace()
False
```

isupper()

La función `isupper()` devuelve un valor booleano `True` o `False` si coincide que la *cadena de caracteres* (página 46) estén en MAYÚSCULAS en cada palabra.

```
>>> 'LEONARDO CABALLERO'.isupper()
True
>>> 'LEONARDO caballero'.isupper()
False
```

lstrip()

La función `lstrip()` devuelve una copia de la *cadena de caracteres* (página 46) con el espacio en blanco inicial eliminado. Si se dan la cadena de caracteres y no es *None* (página 210), elimina los caracteres en la cadena de caracteres en su lugar. Si la cadena de caracteres son `unicode`, serán convertidas a `unicode` antes de eliminar.

```
>>> " leonardo caballero ".lstrip()
'leonardo caballero '
```

lower()

La función `lower()` devuelve una *cadenas de caracteres* (página 46) con MINÚSCULAS en cada palabra.

```
>>> 'LEONARDO CABALLERO'.lower()
'leonardo caballero'
```

ord()

La función `ord()` es el inverso de *chr()* (página 130) dada una cadena representando un carácter Unicode, devuelve el entero del código correspondiente.

```
>>> ord('@')
64
>>> ord('$')
36
>>> ord('^')
94
>>> ord('~')
126
```

replace()

La función `replace()` si encuentra el criterio de la búsqueda de la sub-cadena o la remplaza con la nueva sub-cadena enviado por parámetros en la función.

```
>>> 'leonardo caballero'.replace(" cab", " Cab")
'leonardo Caballero'
```

split()

La función `split()` devuelve una lista con la *cadena de caracteres* (página 46) separada por cada indice de la lista.

```
>>> 'leonardo caballero'.split()
['leonardo', 'caballero']
```

splitlines()

La función `splitlines()` devuelve una lista con la *cadena de caracteres* (página 46) separada por cada salto de linea en cada indice de la lista.

```
>>> 'leonardo jose\ncaballero garcia'.splitlines()
['leonardo jose', 'caballero garcia']
```

startswith()

La función `startswith()` devuelve un valor booleano `True` o `False` si coincide que la cadena inicie con el criterio enviado por parámetros en la función.

```
>>> 'leonardo caballero'.startswith("ca")
False
>>> 'leonardo caballero'.startswith("leo")
True
```

str()

La función `str()` es el constructor del tipo de *cadenas de caracteres* (página 46), se usa crear una *carácter* o *cadenas de caracteres* mediante la misma función `str()`.

Puede convertir un *número entero* (página 41) a una *cadena de caracteres*, de la siguiente forma:

```
>>> str(2)
'2'
```

Puede convertir un *número float* (página 42) a una *cadena de caracteres*, de la siguiente forma:

```
>>> str(2.5)
'2.5'
>>> str(-2.5)
'-2.5'
```

Puede convertir un *número complex* (página 43) a una *cadena de caracteres*, de la siguiente forma:

```
>>> str(2.3+0j)
'(2.3+0j)'
```

Puede convertir un tipo *booleano* (página 44) a una *cadena de caracteres*, de la siguiente forma:

```
>>> str(True)
'True'
>>> str(False)
'False'
```

swapcase()

La función `swapcase()` devuelve una *cadenas de caracteres* (página 46) convertida al opuesto sea MAYÚSCULAS o MINÚSCULAS.

```
>>> 'leonardo caballero'.swapcase()
'LEONARDO CABALLERO'
>>> 'LEONARDO CABALLERO'.swapcase()
'leonardo caballero'
```

title()

La función `title()` devuelve una *cadenas de caracteres* (página 46) con capitales en cada palabra.

```
>>> "leonardo caballero".title()
'Leonardo Caballero'
```

unichr()

La función `unichr()` devuelve una *cadena de caracteres Unicode* de un carácter con un numero entero.

```
>>> unichr(64)
u'@'
>>> unichr(36)
u'$'
>>> unichr(94)
u'^'
>>> unichr(126)
u'~'
```

upper()

La función `upper()` devuelve una *cadenas de caracteres* (página 46) con MAYÚSCULAS en cada palabra.

```
>>> "leonardo caballero".upper()
'LEONARDO CABALLERO'
```

5.6.6 Funciones de secuencias

Las funciones de secuencias se describen a continuación:

all()

La función `all()` toma un contenedor como un argumento. Esta devuelve las funciones integradas `True` si todo los valores en el objeto iterable python tienen un valor de tipo *booleano* (página 44) igual a `True`. Un valor vacío tiene un tipo *booleano* (página 44) igual a `False`.

```
>>> all([' ',' ',' '])
True
>>> all({'*','',''})
False
```

any()

La función `any()` ese toma un argumento y devuelve `True` incluso si, un valor en el objeto iterable tiene un valor de tipo *booleano* (página 44) igual a `True`.

```
>>> any((1,0,0))
True
>>> any((0,0,0))
False
>>> any(range(5))
True
>>> any(range(0))
False
```

coerce()

La función `coerce()` devuelve una tupla que consta de los dos argumentos numéricos convertidos en un tipo común, utilizando las mismas reglas que las operaciones aritméticas. Si la coerción no es posible, levante una excepción *TypeError* (página 193).

```
>>> coerce(3, 4)
(3, 4)
>>> coerce(3, 4.2)
(3.0, 4.2)
```

dict()

La función `dict()` es el constructor del tipo de *diccionario* (página 65), esta función se usa crear un diccionario:

```
>>> dict(python=2.7, zope=2.13, plone=5.1)
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
```

También puede crear un diccionario indicando a las claves usando comillas simples:

```
>>> {'python': 2.7, 'zope': 2.13, 'plone': 5.1}
{'python': 2.7, 'zope': 2, 'plone': 5.1}
>>> dict({'python': 2.7, 'zope': 2.13, 'plone': 5.1})
{{'python': 2.7, 'zope': 2.13, 'plone': 5.1}
```

Convertir desde un grupo de dos *listas* (página 58) de forma recursiva usando la función *zip()* (página 140) a tipo *diccionario*:

```
>>> dict(zip(['python', 'zope', 'plone'], [2.7, 2.13, 5.1]))
{'python': 2.7, 'zope': 2.13, 'plone': 5.1}
```

Convertir desde un grupo de *tuplas* (página 63) respectivamente en una *lista* (página 58) a tipo *diccionario*:

```
>>> dict([('zope', 2.13), ('python', 2.7), ('plone', 5.1)])
{'plone': 5.1, 'zope': 2.13, 'python': 2.7}
```

frozenset()

La función `frozenset()` es el constructor del tipo de *conjuntos* (página 75), se usa crear un conjunto *inmutable* mediante la misma función `frozenset()` de un objeto iterable *lista* (página 58):

```
>>> versiones = [6, 2.1, 2.5, 3.6, 4, 5, 6, 4, 2.5]
>>> print versiones, type(versiones)
[6, 2.1, 2.5, 3.6, 4, 5, 6, 4, 2.5] <type 'list'>
>>> versiones_plone = frozenset(versiones)
>>> print versiones_plone, type(versiones_plone)
frozenset([2.5, 4, 5, 6, 2.1, 3.6]) <type 'frozenset'>
```

iter()

La función `iter()` obtiene un *iterador* (página 92) de un objeto. En la primera forma, el argumento debe proporcionar su propio *iterador*, o ser una secuencia.

```
>>> elemento = iter("Plone")
>>> elemento
<iterator object at 0x7eff6ce10250>
>>> elemento.next()
'P'
>>> elemento.next()
'l'
>>> elemento.next()
'o'
>>> elemento.next()
```

(continué en la próxima página)

(proviene de la página anterior)

```
'n'
>>> elemento.next()
'e'
>>> elemento.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia de tipo *cadena de caracteres* (página 46), al llegar al final mediante el iterador llamado `elemento` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

list()

La función `list()` es el constructor del tipo de *lista* (página 58), se usa crear una lista mediante la misma función `list()` de un iterable. Por ejemplo, una lista podría crearse mediante la función *range(10)* (página 120):

```
>>> lista = list(range(10))
>>> print lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

next()

La función `next()` devuelve el próximo elemento desde un *iterador* (página 92).

```
>>> elemento = iter([1,2,3,4,5])
>>> next(elemento)
1
>>> next(elemento)
2
>>> next(elemento)
3
>>> next(elemento)
4
>>> next(elemento)
5
>>> next(elemento)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior, cuando se itera en la secuencia de tipo *lista* (página 58), al llegar al final mediante el iterador llamado `elemento` se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

tuple()

La función `tuple()` es el constructor del tipo de *tuplas* (página 63), se usa crear una tupla mediante la misma función `tuple()` de un iterable. Por ejemplo, una tupla podría crearse mediante la función *range(10)* (página 120):

```
>>> tupla = tuple(range(4, 9))
>>> print tupla
(4, 5, 6, 7, 8)
```

set()

La función `set()` es el constructor del tipo de *conjuntos* (página 75), se usa crear un conjunto *mutable* mediante la misma función `set()` de un objeto iterable *lista* (página 58):

```
>>> versiones = [2.1, 2.5, 3.6, 4, 5, 6, 4]
>>> print versiones, type(versiones)
[2.1, 2.5, 3.6, 4, 5, 6, 4] <type 'list'>
>>> versiones_plone = set(versiones)
>>> print versiones_plone, type(versiones_plone)
set([2.5, 4, 5, 6, 2.1, 3.6]) <type 'set'>
```

sorted()

La función `sorted()` devuelve una lista ordenada de los elementos del tipo secuencia que recibe como argumento (lista o cadena de caracteres). La secuencia original no es modificada.

```
>>> lista = [23, 13, 7, 37]
>>> sorted(lista)
[7, 13, 23, 37]
```

La función `sorted()` siempre devuelve una lista, aunque reciba como argumento una *cadena de caracteres* (página 46).

```
>>> cadena = "asdlk"
>>> sorted(cadena)
['a', 'd', 'k', 'l', 's']
```

zip()

La función `zip()` devuelve una lista de *tuplas* (página 63), donde cada tupla contiene el elemento i-th desde cada una de los tipos de secuencias de argumento. La lista devuelta es truncada en longitud a la longitud de la secuencia de argumentos más corta.

```
>>> zip(['python', 'zope', 'plone'], [2.7, 2.13, 5.1])
[('python', 2.7), ('zope', 2.13), ('plone', 5.1)]
```

5.6.7 Funciones de objetos

Las funciones de objetos se describen a continuación:

delattr()

La función `delattr()` elimina un atributo con nombre en un objeto; `delattr(x, 'y')` es equivalente a `del x.y`.

```
>>> class Persona:
...     """Clase que representa una Persona"""
...     cedula = "V-13458796"
...     nombre = "Leonardo"
...     apellido = "Caballero"
...     sexo = "M"
...
>>> macagua = Persona()
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> macagua.sexo
'M'
>>> delattr(Persona,'sexo')
>>> macagua.sexo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: Persona instance has no attribute 'sexo'
```

getattr()

La función `getattr()` obtiene un atributo nombrado desde un objeto; de la siguiente forma `getattr(instancia, 'atributo')` el cual es equivalente a `instancia.atributo`. Cuando un argumento predeterminado es dato, es es devuelto cuando el atributo no existe; sin eso, una excepción es lanzada en ese caso.

```
>>> class Persona:
...     """Clase que representa una Persona"""
...     cedula = "V-13458796"
...     nombre = "Leonardo"
...     apellido = "Caballero"
...     sexo = "M"
...
>>> macagua = Persona()
>>> getattr(macagua,'sexo')
'M'
>>> macagua.sexo
'M'
```

hasattr()

La función `hasattr()` devuelve un tipo booleano cuando el objeto tiene un atributo con el nombre dado. (Esta hecho llamando a la función `getattr(instancia, atributo)` y capturar excepciones.)

```
>>> class Persona:
...     """Clase que representa una Persona"""
...     cedula = "V-13458796"
...     nombre = "Leonardo"
...     apellido = "Caballero"
...     sexo = "M"
...
>>> macagua = Persona()
>>> hasattr(macagua, 'nombre')
True
>>> hasattr(macagua, 'apellido')
True
>>> hasattr(macagua, 'cedula')
True
>>> hasattr(macagua, 'sexo')
True
>>> hasattr(macagua, 'email')
False
```

hash()

La función `hash()` devuelve un valor hash de tipo entero para el objeto.

```
>>> class Persona:
...     """Clase que representa una Persona"""
...     cedula = "V-13458796"
...     nombre = "Leonardo"
...     apellido = "Caballero"
...     sexo = "M"
...
>>> macagua = Persona
>>> type(macagua)
<type 'classobj'>
```

Dos objetos con el mismo valor tienen el mismo valor hash.

```
>>> type(Persona)
<type 'classobj'>
>>> type(macagua)
<type 'classobj'>
>>> hash(macagua)
8742669316448
>>> hash(Persona)
8742669316448
```

Lo contrario no es necesariamente cierto, pero es probable.

isinstance()

La función `isinstance()` le permite corroborar si un objeto es una *instancia* (página 201) de una clase.

```
isinstance(objeto, tipo)
```

Esta función devuelve `True` si el objeto especificado es del tipo especificado, de lo contrario `False`.

Los parámetros son:

- *objeto*, es requerido. Un objeto.
- *tipo*, un tipo o una clase, o una tupla de tipos y/o clases

Un ejemplo de uso con la clase `Persona` seria como lo siguiente:

```
>>> persona1 = Persona("V-13458796", "Leonardo", "Caballero", "M")
>>> isinstance(persona1, Persona)
True
```

Si el tipo de parámetro es una tupla, esta función devuelve `True` si le objeto es uno de los tipos en la tupla.

```
>>> persona1 = Persona("V-13458796", "Leonardo", "Caballero", "M")
>>> isinstance(persona1, (Persona, int))
True
```

Aquí puede decir que `persona1` es una instancia de la clase `Persona`.

Las clases dan la posibilidad de crear estructuras de datos más complejas. En el ejemplo, una clase `Persona` que realizará un seguimiento del `cedula`, `nombre`, `apellido` y `sexo` (que pasará como atributos).

issubclass()

La función `issubclass()` le permite corroborar si un objeto es instancia de una clase.

```
issubclass(subclase, clase)
```

Esta función devuelve `True` si la clase especificada es una subclase de la clase base, de lo contrario `False`.

Un ejemplo de uso con la subclase `Supervisor` que deriva de la clase `Persona` seria como lo siguiente:

```
>>> sV1 = Supervisor("V-16987456", "Jen", "Paz", "D", "Chivo")
>>> issubclass(sV1, Persona)
True
```

Si el tipo de parámetro es una tupla, esta función devuelve `True` si le objeto es uno de los tipos en la tupla.

```
>>> sV1 = Supervisor("V-16987456", "Jen", "Paz", "D", "Chivo")
>>> issubclass(sV1, (Persona, Empleado, Supervisor, Destreza))
True
```

Aquí puede decir que `sV1` es una subclase derivada de la clase `Persona`.

Las clases dan la posibilidad de crear estructuras de datos más complejas. En el ejemplo, una clase `Persona` que realizará un seguimiento del `cedula`, `nombre`, `apellido` y `sexo` (que pasará como atributos).

setattr()

La función `setattr()` establecer un atributo con nombre en un objeto; `setattr(x, 'y', v)` es equivalente a `x.y = v`.

```
>>> class Persona:
...     """Clase que representa una Persona"""
...     cedula = "V-13458796"
...     nombre = "Leonardo"
...     apellido = "Caballero"
...     sexo = "M"
...
>>> setattr(macagua, 'email', 'leonardoc@plone.org')
>>> getattr(macagua,'email')
'leonardoc@plone.org'
```

Importante: La lista de todas las funciones disponibles en el lenguaje Python con la descripción correspondiente se puede encontrar en la siguiente dirección URL:

- https://docs.python.org/2/library/functions.html

CAPÍTULO 6

Introspección a la depuración con pdb

En Python puede realizar depuración de programas por defecto usando el módulo `pdb`.

En esta lección se describen como hacer depuración a programas en el lenguaje Python, mostrando ejemplos prácticos y útiles. A continuación el temario de esta lección:

6.1 Depuración con pdb

En este tutorial se exploran herramientas que ayudan a entender tu código: depuración para encontrar y corregir *bugs* (errores).

El depurador python, `pdb`: https://docs.python.org/2/library/pdb.html, te permite inspeccionar tu código de forma interactiva.

Te permite:

- Ver el código fuente.
- Ir hacia arriba y hacia abajo del punto donde se ha producido un error.
- Inspeccionar valores de variables.
- Modificar valores de variables.
- Establecer `breakpoints` (punto de parada del proceso).

print

Sí, las declaraciones `print` sirven como herramienta de depuración. Sin embargo, para inspeccionar en tiempo de ejecución es más eficiente usar el depurador.

6.1.1 Invocando al depurador

Formas de lanzar el depurador:

1. Postmortem, lanza el depurador después de que se hayan producido errores.
2. Lanza el módulo con el depurador.

3. Llama al depurador desde dentro del módulo.

Postmortem

Situación: Estás trabajando en `ipython` y obtienes un error (*traceback*).

En este caso esta depurando el fichero `index_error.py`. Cuando lo ejecutes verás como se lanza una excepción *IndexError* (página 191). Escribe `%debug` y entrarás en el depurador.

```
In [1]: %run index_error.py
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/home/macagua/python/entrenamiento/index_error.py in <module>()
      6
      7 if __name__ == '__main__':
----> 8     index_error()
      9

/home/macagua/python/entrenamiento/index_error.py in index_error()
      3 def index_error():
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6
      7 if __name__ == '__main__':

IndexError: list index out of range

In [2]: %debug
> /home/macagua/python/entrenamiento/index_error.py(5)index_error()
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6

ipdb> list
      1 """Small snippet to raise an IndexError."""
      2
      3 def index_error():
      4     lst = list('foobar')
----> 5     print lst[len(lst)]
      6
      7 if __name__ == '__main__':
      8     index_error()
      9

ipdb> len(lst)
6
ipdb> print lst[len(lst)-1]
r
ipdb> quit

In [3]:
```

Depuración post-mortem sin ipython

En algunas situaciones no podrás usar IPython, por ejemplo para depurar un *script* que ha sido llamado desde la línea de comandos. En este caso, puedes ejecutar el *script* de la siguiente forma `python -m pdb script.py`:

```
python -m pdb index_error.py
> /home/macagua/python/entrenamiento/index_error.py(1)<module>()
-> """Small snippet to raise an IndexError."""
(Pdb) continue
Traceback (most recent call last):
File "/usr/lib/python2.7/pdb.py", line 1296, in main
    pdb._runscript(mainpyfile)
File "/usr/lib/python2.7/pdb.py", line 1215, in _runscript
    self.run(statement)
File "/usr/lib/python2.7/bdb.py", line 372, in run
    exec cmd in globals, locals
File "<string>", line 1, in <module>
File "index_error.py", line 8, in <module>
    index_error()
File "index_error.py", line 5, in index_error
    print lst[len(lst)]
IndexError: list index out of range
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> /home/macagua/python/entrenamiento/index_error.py(5)index_error()
-> print lst[len(lst)]
(Pdb)
```

Ejecución paso a paso

Situación: Crees que existe un error en un módulo pero no estás seguro donde.

Por ejemplo, esta intentado depurar `wiener_filtering.py`. A pesar de que el código se ejecuta, observa que el filtrado no se está haciendo correctamente.

- Ejecuta el *script* en IPython con el depurador usando `%run -d wiener_filtering.py`:

```
In [1]: %run -d wiener_filtering.py
*** Blank or comment
*** Blank or comment
*** Blank or comment
Breakpoint 1 at /home/macagua/python/entrenamiento/wiener_filtering.py:4
NOTE: Enter 'c' at the ipdb>  prompt to start your script.
> <string>(1)<module>()
```

- Coloca un *breakpoint* en la línea 34 usando `b 34`:

```
ipdb> n
> /home/macagua/python/entrenamiento/wiener_filtering.py(4)<module>()
      3
1---> 4 import numpy as np
      5 import scipy as sp

ipdb> b 34
Breakpoint 2 at /home/macagua/python/entrenamiento/wiener_filtering.py:34
```

- Continua la ejecución hasta el siguiente *breakpoint* con `c(ont(inue))`:

```
ipdb> c
> /home/macagua/python/entrenamiento/wiener_filtering.py(34)iterated_wiener()
     33     """
2--> 34     noisy_img = noisy_img
     35     denoised_img = local_mean(noisy_img, size=size)
```

- Da pasos hacia adelante y detrás del código con `n(ext)` y `s(tep)`. `next` salta hasta la siguiente declaración en el actual contexto de ejecución mientras que `step` se moverá entre los contextos en ejecución, i.e.

permitiendo explorar dentro de llamadas a funciones:

```
ipdb> s
> /home/macagua/python/entrenamiento/wiener_filtering.py(35)iterated_wiener()
2     34     noisy_img = noisy_img
---> 35     denoised_img = local_mean(noisy_img, size=size)
     36     l_var = local_var(noisy_img, size=size)

ipdb> n
> /home/macagua/python/entrenamiento/wiener_filtering.py(36)iterated_wiener()
     35     denoised_img = local_mean(noisy_img, size=size)
---> 36     l_var = local_var(noisy_img, size=size)
     37     for i in range(3):
```

- Muévete unas pocas líneas y explora las variables locales:

```
ipdb> n
> /home/macagua/python/entrenamiento/wiener_filtering.py(37)iterated_wiener()
     36     l_var = local_var(noisy_img, size=size)
---> 37     for i in range(3):
     38         res = noisy_img - denoised_img
ipdb> print l_var
[[5868 5379 5316 ..., 5071 4799 5149]
 [5013  363  437 ...,  346  262 4355]
 [5379  410  344 ...,  392  604 3377]
 ...,
 [ 435  362  308 ...,  275  198 1632]
 [ 548  392  290 ...,  248  263 1653]
 [ 466  789  736 ..., 1835 1725 1940]]
ipdb> print l_var.min()
0
```

Oh dear, solo ve entero y variación 0. Aquí está nuestro error, estamos haciendo aritmética con enteros.

Lanzando excepciones en errores numéricos

Cuando ejecuta el fichero `wiener_filtering.py`, se lanzarán los siguientes avisos:

```
In [2]: %run wiener_filtering.py
wiener_filtering.py:40: RuntimeWarning: divide by zero encountered in divide
    noise_level = (1 - noise/l_var )
```

Puede convertir estos avisos a excepciones, lo que le permitiría hacer una depuración post-mortem sobre ellos y encontrar el problema de manera más rápida:

```
In [3]: np.seterr(all='raise')
Out[3]: {'divide': 'print', 'invalid': 'print', 'over': 'print', 'under': 'ignore
↪'}
In [4]: %run wiener_filtering.py
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
/home/macagua/venv/lib/python2.7/site-packages/IPython/utils/py3compat.pyc
in execfile(fname, *where)
    176             else:
    177                 filename = fname
--> 178             __builtin__.execfile(filename, *where)

/home/macagua/python/entrenamiento/wiener_filtering.py in <module>()
     55 pl.matshow(noisy_lena[cut], cmap=pl.cm.gray)
     56
---> 57 denoised_lena = iterated_wiener(noisy_lena)
     58 pl.matshow(denoised_lena[cut], cmap=pl.cm.gray)
     59

/home/macagua/python/entrenamiento/wiener_filtering.py in
iterated_wiener(noisy_img, size)
     38         res = noisy_img - denoised_img
     39         noise = (res**2).sum()/res.size
---> 40         noise_level = (1 - noise/l_var )
     41         noise_level[noise_level<0] = 0
```

Otras formas de comenzar una depuración

- **Lanzar una excepción «break point» a lo pobre**

  Si encuentras tedioso el tener que anotar el número de línea para colocar un *break point*, puedes lanzar una excepción en el punto que quieres inspeccionar y usar la "magia" `%debug` de `ipython`. Destacar que en este caso no puedes moverte por el código y continuar después la ejecución.

- **Depurando fallos de pruebas usando nosetests**

  Puede ejecutar `nosetests --pdb` para saltar a la depuración post-mortem de excepciones y `nosetests --pdb-failure` para inspeccionar los fallos de pruebas usando el depurador.

  Además, puedes usar la interfaz IPython para el depurador en **nose** usando el plugin de **nose** ipdbplugin[62]. Puede, entonces, pasar las opciones `--ipdb` y `--ipdb-failure` a los *nosetests*.

- **Llamando explícitamente al depurador**

  Inserta la siguiente línea donde quieres que salte el depurador:

```
import pdb; pdb.set_trace()
```

Advertencia: Cuandos e ejecutan `nosetests`, se captura la salida y parecerá que el depurador no está funcionando. Para evitar esto simplemente ejecuta los `nosetests` con la etiqueta `-s`.

Depuradores gráficos y alternativas

- Quizá encuentres más conveniente usar un depurador gráfico como winpdb[63]. para inspeccionar saltas a través del código e inspeccionar las variables
- De forma alternativa, pudb[64] es un buen depurador semi-gráfico con una interfaz de texto en la consola.
- También, estaría bien echarle un ojo al proyecto pydbgr[65]

6.1.2 Comandos del depurador e interacciones

`l(list)`	Lista el código en la posición actual
`u(p)`	Paso arriba de la llamada a la pila (*call stack*)
`d(own)`	Paso abajo de la llamada a la pila ((*call stack*)
`n(ext)`	Ejecuta la siguiente línea (no va hacia abajo en funciones nuevas)
`s(tep)`	Ejecuta la siguiente declaración (va hacia abajo en las nuevas funciones)
`bt`	Muestra el *call stack*
`a`	Muestra las variables locales
`!command`	Ejecuta el comando **Python** proporcionado (en oposición a comandos pdb)

[62] https://pypi.org/project/ipdbplugin
[63] https://pypi.org/project/winpdb/
[64] https://pypi.org/project/pudb
[65] https://code.google.com/archive/p/pydbgr

Advertencia: Los comandos de depuración no son código Python

No puedes nombrar a las variables de la forma que quieras. Por ejemplo, si esta dentro del depurador no podrá sobrescribir a las variables con el mismo y, por tanto, **habrá que usar diferentes nombres para las variables cuando este tecleando código en el depurador**.

6.1.3 Obteniendo ayuda dentro del depurador

Teclea `h` o `help` para acceder a la ayuda interactiva:

```
ipdb> help

Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pdef   r        tbreak   w
a      c          continue  exit    l     pdoc   restart  u        whatis
alias  cl         d         h       list  pinfo  return   unalias  where
args   clear      debug     help    n     pp     run      unt
b      commands   disable   ignore  next  q      s        until
break  condition  down      j       p     quit   step     up

Miscellaneous help topics:
==========================
exec  pdb

Undocumented commands:
======================
retval  rv
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic en los siguientes enlaces: `index_error.py` y `wiener_filtering.py`. Adicional se incluye otro código de ejemplo muy simple `funcion_a_depurar.py` usando la función `set_trace()` del paquete `pdb`.

Truco: Para ejecutar el código `index_error.py`, `wiener_filtering.py` y `funcion_a_depurar.py`, abra una consola de comando, acceda al directorio donde se encuentra ambos programas:

```
leccion6/
├── index_error.py
├── wiener_filtering.py
└── funcion_a_depurar.py
```

Si tiene la estructura de archivo previa, entonces ejecute por separado cada comando:

```
python index_error.py
python wiener_filtering.py
python funcion_a_depurar.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 264) del entrenamiento para ampliar su conocimiento en esta temática.

CAPÍTULO 7

Operaciones de E/S y manipulación de archivos

En Python puede realizar operaciones de entrada y salida de datos, almacenar esos datos manipulando de archivos usando métodos para leer y escribir archivos desde el sistema de archivos.

En esta lección se describen las operaciones de entrada y salida, manipulando de archivos del lenguaje Python, mostrando ejemplos prácticos y útiles. A continuación el temario de esta lección:

7.1 Entrada/Salida en Python

Los programas serían de muy poca utilidad si no fueran capaces de interaccionar con el usuario.

7.1.1 Entrada estándar

Para pedir información al usuario, debe utilizar las funciones integradas en el interprete del lenguaje, así como los argumentos de línea de comandos.

Ejemplo de la función raw_input:

La función *raw_input()* (página 123) siempre devuelve un valor de cadenas de caracteres:

```
>>> nombre = raw_input('Ana: ¿Cómo se llama usted?: ')
Ana: ¿Cómo se llama usted?: Leonardo
>>> print nombre
Leonardo
```

Ejemplo de la función input:

La función *input()* (página 123) siempre devuelve un valor numérico:

```
>>> edad = input('Ana: ¿Que edad tiene usted?: ')
Ana: ¿Que edad tiene usted?: 38
>>> print edad
38
```

7.1.2 Entrada por script

En muchas practicas de este entrenamiento usted lo que ha hecho ha sido escribir código en el intérprete, y/o escribir/ejecutar pequeños programas Python, pero los programas informáticos no funcionan así.

Se basan en escribir todas las instrucciones en archivos llamados *scripts*, que no es mas que guiones de instrucciones. Luego se envía este archivo al intérprete como parámetro desde la terminal de comando (si es un lenguaje interpretado como Python) y éste ejecutará todas las instrucciones en bloque.

A parte de ser la base del funcionamiento de los programas, la característica de los *scripts* es que pueden recibir datos desde la propia terminal de comando en el momento de la ejecución, algo muy útil para agregar dinamismo los *scripts* a través de parámetros personalizables.

A continuación, un ejemplo el cual simula a sala de chat del servicio *LatinChat.com*, validando datos de entradas numérico y tipo cadena de caracteres e interactuá con el usuario y en base a condicionales muestra un mensaje.

```
print "\nSimulando a LatinChat"
print "====================="

print "\nSala de Chat > De 30 a 40 años"
print "------------------------------\n"

print 'Ana: ¿Cómo se llama usted?: '
nombre = raw_input('Yo: ')
print 'Ana: Hola', nombre, ', encantada de conocerte :3'

print 'Ana: ¿Que edad tiene usted?: '
edad = input('Yo: ')
print 'Usted tiene', edad, ', y yo ya no digo mi edad xD'

print 'Ana: ¿Tiene WebCam?, ingrese "si" o "no", por favor!: '
tiene_WebCam = raw_input('Yo: ')

if tiene_WebCam in ('s', 'S', 'si', 'Si', 'SI'):
        print "Ponga la WebCam para verle :-D"
elif tiene_WebCam in ('n', 'no', 'No', 'NO'):
        print "Lastima por usted :'( Adiós"
```

Truco: LatinChat.com, fue un servicio de Internet que ofrecía diversas salas de chat, muy popular en la década de los 90 en latinoamericana.

Scripts con argumentos

Para poder enviar información a un script y manejarla, tenemos que utilizar la librería de sistema `sys`. En ella encontraremos la lista `argv` que almacena los argumentos enviados al *script*.

Usted debe crear un *script* llamado `entrada_argumentos.py` con el siguiente contenido:

```
import sys

print sys.argv
```

Ejecuta el *script* llamado `entrada_argumentos.py`, de la siguiente forma:

```
python entrada_argumentos.py
['entrada_argumentos.py']
```

Al ejecutarlo puede ver que devuelve una lista con una cadena que contiene el nombre del *script*. Entonces, el primer argumento de la lista `sys.argv` (es decir, `sys.argv[0]`) es el propio nombre del *script*.

Ahora si intenta ejecutar el *script* de nuevo pasando algunos valores como números y cadenas de caracteres entre comillas dobles, todo separado por espacios:

```
python entrada_argumentos.py 300 43.234 "Hola Plone"
['entrada_argumentos.py', '300', '43.234', 'Hola Plone']
```

Cada valor que enviamos al *script* durante la llamada se llama argumento e implica una forma de entrada de datos alternativa sin usar las funciones *input()* (página 123) y *raw_input()* (página 123).

A continuación, un ejemplo el cual usa un *script* con la librería `sys`. El *script* recibe dos (02) argumentos: una cadena de caracteres y un número entero. Lo que hace es imprimir la cadena de caracteres tantas veces como le indique con el argumento de tipo número:

```
import sys

# Comprobación de seguridad, ejecutar sólo si se reciben 2
# argumentos realemente
if len(sys.argv) == 3:
    texto = sys.argv[1]
    repeticiones = int(sys.argv[2])
    for r in range(repeticiones):
        print texto
else:
    print "ERROR: Introdujo uno (1) o más de dos (2) argumentos"
    print "SOLUCIÓN: Introduce los argumentos correctamente"
    print 'Ejemplo: entrada_dos_argumentos.py "Texto" 5'
```

Si quiere comprobar la validación de cuantos argumentos deben enviarme al script, ejecute el siguiente comando:

```
python entrada_dos_argumentos.py "Hola Plone"
ERROR: Introdujo uno (1) o más de dos (2) argumentos
SOLUCIÓN: Introduce los argumentos correctamente
Ejemplo: entrada_dos_argumentos.py "Texto" 5
```

Ahora si intenta ejecutar el *script* `entrada_dos_argumentos.py` con solo dos (2) argumentos, ejecutando el siguiente comando:

```
python entrada_dos_argumentos.py "Hola Plone" 3
Hola Plone
Hola Plone
Hola Plone
```

7.1.3 Salida estándar

La forma general de mostrar información por pantalla es mediante una consola de comando, generalmente podemos mostrar texto y variables separándolos con comas, para este se usa la sentencia *print* (página 153).

Sentencia print

Sentencia `print` evalúa cada expresión, devuelve y escribe el objeto resultado a la salida estándar de la consola de comando. Si un objeto no es un *tipo cadena de caracteres* (página 46), ese es primeramente convertido a un *tipo cadena de caracteres* usando las reglas para las *conversiones del tipo* (página 136). La *cadena de caracteres* (resultado o original) es entonces escrito.

Entonces para mostrar mensajes en pantalla, se utiliza el uso de la sentencia `print`.

Ejemplo del uso de la sentencia print:

```
>>> print 'Ana: Hola', nombre, ', encantada de conocerte :3'
Ana: Hola Leonardo , encantado de conocerte :3
```

Formato de impresión de cadenas

En la sentencia `print` se pueden usar el formato de impresión alternando las cadenas de caracteres y variables:

```
>>> tipo_calculo = "raíz cuadrada de dos"
>>> valor = 2**0.5
>>> print "el resultado de", tipo_calculo, "es:", valor
el resultado de raíz cuadrada de dos es: 1.41421356237
```

Ver también:

Hay disponibles otras formas de aplicar formato de cadenas de caracteres:

- *Formateo %* (página 53).
- *Clase formatter* (página 53).

Nota: Una documentación completa del control de la salida de Python se encuentra en https://docs.python.org/2/tutorial/inputoutput.html

---

Importante: Usted puede descargar los códigos usados en esta sección haciendo clic en los siguientes enlaces: `entrada_salida.py`, `entrada_argumentos.py` y `entrada_dos_argumentos.py`.

Truco: Para ejecutar el código `entrada_salida.py`, `entrada_argumentos.py` y `entrada_dos_argumentos.py`, abra una consola de comando, acceda al directorio donde se encuentra ambos programas:

```
leccion8/
├── entrada_argumentos.py
├── entrada_dos_argumentos.py
└── entrada_salida.py
```

Si tiene la estructura de archivo previa, entonces ejecute el siguiente comando:

```
python entrada_salida.py
python entrada_argumentos.py
python entrada_dos_argumentos.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

7.2 Manipulación de archivos

Para escribir o leer cadenas de caracteres para/desde archivos (otros tipos deben ser convertidas a cadenas de caracteres). Para esto Python incorpora un tipo integrado llamado *file* (página 214), el cual es manipulado mediante un objeto archivo el cual fue generado a través de una función integrada en Python, a continuación se describen los procesos típicos y sus referencias a funciones propias del lenguaje:

7.2.1 Abrir archivo

La forma preferida para abrir un archivo es usando la función integrada *open()* (página 119).

7.2.2 Leer archivo

La forma preferida para leer un archivo es usando algunas de los métodos del tipo objeto *file* (página 214) como *read()* (página 216), *readline()* (página 217) y *readlines()* (página 217).

7.2.3 Escribir archivo

La forma preferida para escribir un archivo es usando el método del tipo objeto *file* (página 214) llamado *write()* (página 219).

7.2.4 Cerrar archivo

La forma preferida para cerrar un archivo es usando el método del tipo objeto *file* (página 214) llamado *close()* (página 215).

7.2.5 Archivos con modulo os

El módulo `os` de Python le permite a usted realizar operaciones dependiente del *Sistema Operativo* como crear una carpeta, listar contenidos de una carpeta, conocer acerca de un proceso, finalizar un proceso, etc. Este módulo tiene métodos para ver variables de entornos del *Sistema Operativo* con las cuales Python esta trabajando en mucho más. Aquí[66] la documentación Python para el módulo `os`.

A continuación algunos útiles métodos del módulo `os` que pueden ayudar a manipular archivos y carpeta en su programa Python:

Crear una nueva carpeta

```
>>> import os
>>> os.makedirs("Ana_Poleo")
```

Listar el contenidos de una carpeta

```
>>> import os
>>> os.listdir("./")
['Ana_Poleo']
```

Mostrar el actual directorio de trabajo

```
>>> import os
>>> os.getcwd()
'/home/usuario/python/'
```

Mostrar el tamaño del archivo en `bytes` del archivo pasado en parámetro

```
>>> import os
>>> os.path.getsize("Ana_Poleo")
4096
```

¿Es un archivo el parámetro pasado?

```
>>> import os
>>> os.path.isfile("Ana_Poleo")
False
```

¿Es una carpeta el parámetro pasado?

[66] https://docs.python.org/2/library/os.html

```
>>> import os
>>> os.path.isdir("Ana_Poleo")
True
```

Cambiar directorio/carpeta

```
>>> import os
>>> os.chdir("Ana_Poleo")
>>> os.getcwd()
'/home/usuario/python/Ana_Poleo'
>>> os.listdir("./")
[]
>>> os.chdir("../")
>>> os.getcwd()
'/home/usuario/python'
```

Renombrar un archivo

```
>>> import os
>>> os.rename("Ana_Poleo","Ana_Carolina")
>>> os.listdir("./")
['Ana_Carolina']
```

Eliminar un archivo

```
>>> import os
>>> os.chdir("Ana_Carolina")
>>> archivo = open(os.getcwd()+'/datos.txt', 'w')
>>> archivo.write("Se Feliz!")
>>> archivo.close()
>>> os.getcwd()
'/home/usuario/python/Ana_Carolina'
>>> os.listdir("./")
['datos.txt']
>>> os.remove(os.getcwd()+"/datos.txt")
>>> os.listdir("./")
[]
```

Eliminar una carpeta

```
>>> os.rmdir("Ana_Carolina")
>>> os.chdir("Ana_Carolina")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OSError: [Errno 2] No such file or directory: 'Ana_Carolina'
```

Lanza una excepción *OSError* (página 192) cuando intenta acceder al directorio que previamente elimino y este no encuentra.

7.2.6 Ejemplos de archivos

A continuación, se presentan algunos ejemplos del uso del tipo objeto *file* (página 214):

Ejemplo de iterar un archivo para leerlo

Usted puede iterar sobre un archivo como se muestra a continuación:

```
>>> archivo = open('datos.txt', 'r')
>>> for linea in archivo:
...     print linea
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
Este es una prueba

y otra prueba
>>> archivo.close()
```

Ejemplo de iterar un archivo con escritura y lectura

Usted puede manipular un archivo con permisos de escritura y lectura, ademas de interactuar de el mismo como se muestra a continuación:

```
import os

print "\nCrear un archivo"
print "================"

NOMBRE_ARCHIVO = 'datos.txt'

archivo = open(NOMBRE_ARCHIVO, 'w') # abre el archivo datos.txt
archivo.write('Este es una prueba \ny otra prueba.')
archivo.close()

if NOMBRE_ARCHIVO in os.listdir("."):
    print "\nArchivo creado en la ruta: \n\n\t{0}/{1}".format(
        os.getcwd(), NOMBRE_ARCHIVO)
else:
    print "El archivo no fue creado!!!\n"


print "\n\nLeer un archivo"
print "===============\n"

archivo = open(NOMBRE_ARCHIVO, 'r')
contenido = archivo.read()
print contenido
archivo.close()


print "\n\nIterar sobre un archivo"
print "======================\n"

archivo = open(NOMBRE_ARCHIVO, 'r')
for linea in archivo:
    print linea
print "\n"
archivo.close()


print "\nEliminar un archivo"
print "==================="

os.remove(os.getcwd()+"/"+NOMBRE_ARCHIVO)
print "\nEliminado archivo desde la ruta: \n\n\t{0}/{1}".format(
    os.getcwd(), NOMBRE_ARCHIVO)
```

7.2.7 Ayuda integrada

Usted puede consultar toda la documentación disponible sobre los tipos objeto *file* (página 214) desde la *consola interactiva* (página 15) de la siguiente forma:

```
>>> help(file)
```

Para salir de esa ayuda presione la tecla `q`.

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `archivo.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python archivo.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

CAPÍTULO 8

Módulos, paquetes y distribución de software

En Python las diversas aplicaciones Python se encuentran dentro de módulos y paquetes los cuales los contienen el sistema de ficheros.

En esta lección se describen como crear módulos y paquetes, luego se enseña a usar las practicas de scaffolding de paquetes Python con mecanismo de instalación, mostrando ejemplos prácticos y útiles. A continuación el temario de esta lección:

8.1 Módulos Python

Un módulo le permite a usted organizar lógicamente su código Python. Agrupando código relacionado dentro de un módulo hace el código mas fácil de entender y usar. Un módulo es un objeto de Python con atributos con nombres arbitrarios que puede enlazar y hacer referencia.

Simplemente, un módulo es no es otra cosa sino un archivo con extensión **.py**. Un módulo puede definir funciones, clases y variables, también puede incluir código ejecutable.

El código Python para un módulo nombrado `funciones` normalmente reside un archivo llamado `utilidades.py`. A continuación un ejemplo de un simple módulo llamado `utilidades.py`:

```
""" Módulo para cálculos diversos """

def suma_total(monto=0):
    """ Calcula la suma total """
    calculo_suma = 20
    calculo_suma += monto
    return calculo_suma
```

8.1.1 Sentencia import

La sentencia `import` se utiliza para importar un módulo. Usted puede usar cualquier archivo de código Python como un módulo ejecutando esta sentencia en otro archivo de código Python. La sentencia `import` tiene la siguiente sintaxis:

```
>>> import os
>>> import re, datetime
```

Cuando el interprete encuentra una sentencia `import`, este importa el módulo si el mismo esta presente en la ruta de búsqueda. Una ruta de búsqueda es una lista de directorios que el interprete busca antes de importar un módulo.

Por ejemplo, al importar el módulo `utilidades.py`, usted necesita colocar la siguiente sentencia al tope del otro script Python. A continuación un ejemplo de un simple módulo, `calculo_factura_pipo.py`.

```
# Importar el modulo llamado "utilidades"
import utilidades

print "Importo el modulo '{0}'\n".format(
    utilidades.__file__.replace(
            utilidades.__file__, "utilidades.pyc"))

print u"Función '{0}' del módulo '{1}' llamado y mostró:".format(
    utilidades.suma_total.__name__,
    utilidades.__file__.replace(
            utilidades.__file__, "utilidades.pyc"))

# Usted puede llamar una función definida dentro del módulo
print "Monto total a facturar: {0} BsS.".format(
        utilidades.suma_total(int(input("Ingrese un monto: "))))
```

Cuando el código anterior es ejecutado, ese produce el siguiente resultado:

```
Importo el modulo 'utilidades.pyc'

Función 'suma_total' del módulo 'utilidades.pyc' llamado y mostró:
Ingrese un monto: 56987
Monto total a facturar: 57007 BsS.
```

Un módulo se carga solo una vez, independientemente de la cantidad de veces que se importe. Esto evita que la ejecución del módulo ocurra una y otra vez si se producen múltiples importaciones.

La primera vez que un módulo es importado en un script de Python, se ejecuta su código una vez. Si otro módulo importa el mismo módulo este no se cargará nuevamente; los módulos son inicializados una sola vez.

Esto se debe al código objeto compilado que genera en el mismo directorio del módulo que cargo con la extensión de archivo **.pyc**, ejecutando las siguientes sentencias:

```
>>> import funciones, os
>>> archivos = os.listdir(os.path.abspath(
...     funciones.__file__).replace("/utilidades.pyc", "/"))
>>> print filter(lambda x: x.startswith('funciones.'), archivos)
['utilidades.py', 'utilidades.pyc']
```

De esta forma se comprueba que existe el archivo compilado de Python junto con el mismo módulo Python.

8.1.2 Localizando módulos

Cuando usted importa un módulo, el interprete Python busca por el módulo en la secuencia siguiente:

1. El directorio actual.
2. Si el módulo no es encontrado, Python entonces busca en cada directorio en la variable de entorno *PYTHONPATH* (página 161) del sistema operativo.
3. Si todas las anteriores fallan, Python busca la ruta predeterminada. En UNIX, la ruta predeterminada normalmente esta `/usr/local/lib/python/`.

El ruta de búsqueda de módulo es almacenado en el módulo de system `sys` como la variable `sys.path`. La variable `sys.path` contiene el directorio actual, `PYTHONPATH`, y las predeterminadas dependencia de instalación.

8.1.3 PYTHONPATH

Es una variable de entorno del sistema operativo, consistiendo de una lista de directorios. La sintaxis de `PYTHONPATH` es la misma como la del shell de la variable `PATH`.

Así es una típica definición de `PYTHONPATH` desde un sistema Windows, ejecutando:

```
set PYTHONPATH = C:\python20\lib;
```

Así es una típica definición de `PYTHONPATH` desde un sistema UNIX, ejecutando:

```
set PYTHONPATH = /usr/local/lib/python
```

8.1.4 Espacios de nombres y alcance

Las *variables* (página 26) son nombres (identificadores) que se asignan a objetos.

Un espacio de nombres o namespace, es un diccionario de nombres de variables (claves) y sus objetos (valores) correspondientes.

Una sentencia de Python puede acceder a las variables en un espacio de nombres local y en el espacio de nombres global. Si una *variable local* y una *variable global* tienen el mismo nombre, la *variable local* sombrea la *variable global*.

Cada *función* (página 100) tiene su propio espacio de nombres local. Los *métodos* (página 198) de Clase siguen la misma regla de alcance que las funciones ordinarias.

Python hace conjeturas educadas sobre si las variables son locales o globales. Se supone que cualquier variable asignada a un valor en una función es local.

Por lo tanto, para asignar un valor a una variable global dentro de una función, primero debe usar la sentencia *global* (página 31).

```
>>> global nombre
>>> nombre
'Leonardo'
```

La sintaxis `global nombre`, le dice al interprete Python que la variable `nombre` es una *variable global*. Python deja de buscar la variable en el espacio de nombres local.

Por ejemplo, defina una variable `Money` en el espacio de nombres global. Dentro de la función `Money`, asigna un valor a `Money`, por lo tanto, Python asume que `Money` es una variable local. Sin embargo, accede al valor de la variable local `Money` antes de configurarlo, por lo que el resultado es una excepción *UnboundLocalError* (página 193). Si descomenta la sentencia `global`, se soluciona el problema.

Importante: Usted puede descargar el código usado en esta sección haciendo clic en los siguientes enlaces: `utilidades.py` y `calculo_factura_pipo.py`.

Truco: Para ejecutar el código `utilidades.py` y `calculo_factura_pipo.py`, abra una consola de comando, acceda al directorio donde se encuentra ambos programas:

```
leccion8/
├── utilidades.py
└── calculo_factura_pipo.py
```

Si tiene la estructura de archivo previa, entonces ejecute el siguiente comando:

```
python calculo_factura_pipo.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

8.2 Paquetes Python

Los paquetes pueden contener módulos y otros paquetes. Son directorios. El único requisito es que contengan un archivo llamado `__init__.py`. Este archivo puede estar vacío.

8.2.1 Sentencia from

La sentencia `from` se utiliza en conjunto a la previa sentencia *import* (página 159) para importar un módulo.

```
>>> from utilidades import suma_total
```

Por ejemplo, cree un directorio llamado `tostadas_pipo`, que contiene los archivos llamados `__init__.py`, `principal.py` (dentro del mismo directorio).

- Archivo `__init__.py`, este archivo no tiene ningún contenido.
- Archivo `principal.py` incluye el siguiente código:

```
from utilidades import impuestos
from utilidades import calculos

monto = int(input("Introduzca un monto entero: "))
# Llama función definida en el módulo "impuestos"
print "El impuesto IVA de 12%:", impuestos.impuesto_iva12(monto)

suma = int(input("Introduzca un monto entero a sumar: "))
# Llama función definida en el módulo "calculos"
print "La suma total es:", calculos.suma_total(suma)
```

Seguidamente dentro del directorio `tostadas_pipo`, cree otro directorio llamado `utilidades`, dentro de este, cree los siguientes archivos:

- Archivo `__init__.py`, este archivo no tiene ningún contenido.
- Archivo `calculos.py` incluye el siguiente código:

```
""" Módulo para cálculos diversos """

def suma_total(monto=0):
    """ Calcula la suma total """
    calculo_suma = 20
    calculo_suma += monto
    return calculo_suma
```

- Archivo `impuestos.py` incluye el siguiente código:

```
""" Módulo para cálculos de diversos impuestos """

def impuesto_iva12(monto=0):
    """ Calcula el impuesto del IVA de 12 % """
    total = ((monto * 12)/100)
    return total
```

(continué en la próxima página)

(proviene de la página anterior)

```
def impuesto_iva14(monto=0):
    """ Calcula el impuesto del IVA de 14 % """
    total = ((monto * 14)/100)
    return total
```

Al final tendrá la siguiente estructura del directorios del paquete Python llamado `tostadas_pipo`, como se describe a continuación:

```
tostadas_pipo/
├── __init__.py
├── principal.py
└── utilidades/
    ├── calculos.py
    ├── impuestos.py
    └── __init__.py
```

Entonces realizar importaciones desde una estructura de directorios mas completa se realiza de las siguientes formas:

- Importar todos los módulo el sub-paquete `utilidades`, ejecutando:

```
import tostadas_pipo.utilidades
from tostadas_pipo import utilidades
from tostadas_pipo.utilidades import *
```

- Importar el módulo `calculos.py` desde el sub-paquete `utilidades`, ejecutando:

```
from tostadas_pipo.utilidades import calculos
```

- Importar la función `impuesto_iva14()` desde el módulo `impuestos.py` en el sub-paquete `utilidades`, ejecutando:

```
from tostadas_pipo.utilidades.impuestos import impuesto_iva14
```

Por ejemplo, cree un módulo llamado `calculo_factura_pipo.py`, que contiene las importaciones del paquete `tostadas_pipo`:

- Archivo `calculo_factura_pipo.py` incluye el siguiente código:

```
from tostadas_pipo.utilidades import calculos
from tostadas_pipo.utilidades.impuestos import impuesto_iva14

monto = int(input("Introduzca un monto entero: "))
monto_suma = int(input("Introduzca un monto entero a sumar: "))

suma = impuesto_iva14(monto) + calculos.suma_total(monto_suma)

print "Total a Facturar: {0} BsS, con IVA 14%.".format(suma)
```

Importante: Usted puede descargar el código usado en esta sección, haciendo clic en el siguiente enlace: `paquetes.zip`.

Truco: Para ejecutar el código incluido en el archivo `paquetes.zip` debe descomprimirlo, abra una consola de comando, acceda al directorio donde se encuentra el archivo descomprimido, de la siguiente forma:

```
calculo_factura_pipo.py
tostadas_pipo/
├── __init__.py
├── principal.py
└── utilidades/
    ├── calculos.py
    ├── impuestos.py
    └── __init__.py
```

Si tiene la estructura de archivo previa, entonces ejecute el siguiente comando:

```
python calculo_factura_pipo.py
python tostadas_pipo/principal.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

8.3 Distribución de Software

La distribución de código Python, le permite hacer portable de forma amigable usando herramienta de gestión de paquetes Python como la herramienta `pip`. Esta labor se hace mediante el módulo *distutils* (página 164), y más reciente incorporando el módulo *setuptools* (página 164).

8.3.1 Módulo distutils

Permite «empacar» el código de un proyecto de software para ser redistribuido en otros proyectos Python.

Cada paquete empaquetado se puede distribuir en su propia pagina de proyecto y al mismo tiempo puede optar a publicar su proyecto en el Python Package Index (PyPI), con el cual si lo publica allí su proyecto estará a su alcance y sino de muchos mas programadores, ya que es un repositorio de software publico, solo con ejecutar el comando `pip install <paquete>` lo convierte en una herramienta tremendamente útil y probablemente sea una de las razones del éxito de Python entre los que empiezan a programar.

8.3.2 Módulo setuptools

El módulo `setuptools`, incorpora varias extensiones al módulo `distutils` para distribuciones de software grandes o complejas.

8.3.3 Estructura de proyecto

Para poder empaquetar un proyecto necesita como mínimo la estructura de archivos siguiente:

```
DIRECTORIO-DEL-PROYECTO
├── LICENSE
├── MANIFEST.in
├── README.txt
├── setup.py
└── NOMBRE-DEL-PAQUETE
    ├── __init__.py
    ├── ARCHIVO1.py
    ├── ARCHIVO2.py
```

(continué en la próxima página)

(proviene de la página anterior)

```
    └── MODULO (OPCIONAL)
        ├── __init__.py
        └── MAS_ARCHIVOS.py
```

A continuación se detallan el significado y uso de la estructura de directorio anterior:

- `DIRECTORIO-DEL-PROYECTO` puede ser cualquiera, no afecta en absoluto, lo que cuenta es lo que hay dentro.
- `NOMBRE-DEL-PAQUETE` tiene que ser el nombre del paquete, si el nombre es `tostadas_pipo`, este directorio tiene que llamarse también `tostadas_pipo`. Y esto es así. Dentro estarán todos los archivos que forman la librería.
- `LICENSE`: es el archivo donde se define los términos de licencia usado en su proyecto. Es muy importate que cada paquete cargado a PyPI incluirle una copia de los términos de licencia. Esto le dice a los usuario quien instala el paquete los términos bajos los cuales pueden usarlo en su paquete. Para ayuda a seleccionar una licencia, consulte https://choosealicense.com/. Una vez tenga seleccionado una licencia abra el archivo `LICENSE` e ingrese el texto de la licencia. Por ejemplo, si usted elije la licencia GPL:

```
License
=======

PACKAGE-NAME Copyright YEAR, PACKAGE-AUTHOR

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston,
MA 02111-1307 USA.
```

- `MANIFEST.in`: es el archivo donde se define los criterios de inclusión y exclusión de archivos a su distribución de código fuente de su proyecto. Este archivo incluye la configuración del paquete como se indica a continuación:

```
include LICENSE
include *.txt *.in
include *.py
recursive-include tostadas_pipo *
global-exclude *.pyc *.pyo *~
prune build
prune dist
```

- `README.txt`: es el archivo donde se define la documentación general del paquete, este archivo es importante debido a que no solo es usado localmente en un copia descargada, sino como información usada el en sitio de PyPI. Entonces abra el archivo `README.txt` e ingrese el siguiente contenido. Usted puede personalizarlo como quiera:

```
==================
NOMBRE-DEL-PAQUETE
==================

Este es un ejemplo simple de un paquete Python.
```

(continué en la próxima página)

(proviene de la página anterior)

```
Usted puede usar para escribir este contenido la guía
`Restructured Text (reST) and Sphinx CheatSheet <http://openalea.gforge.inria.
↪fr/doc/openalea/doc/_build/html/source/sphinx/rest_syntax.html>`_.
```

- `setup.py`: es el archivo donde se define el paquete, el formato es el mismo para el módulo *setuptools* (página 164) y para el módulo *distutils* (página 164). Lo puede ver a continuación. Este archivo incluye la configuración del paquete como se indica a continuación:

```
"""Instalador para el paquete "tostadas_pipo"."""

from setuptools import setup

long_description = (
    open('README.txt').read()
    + '\n' +
    open('LICENSE').read()
    + '\n')

setup(
    name="tostadas_pipo",
    version="0.1",
    description="Sistema Administrativo de Tostadas Pipo C.A.",
    long_description=long_description,
    # Get more https://pypi.org/pypi?%3Aaction=list_classifiers
    classifiers=[
        # ¿Cuan maduro esta este proyecto? Valores comunes son
        #   3 - Alpha
        #   4 - Beta
        #   5 - Production/Stable
        "Development Status :: 3 - Alpha",
        # Indique a quien va dirigido su proyecto
        "Environment :: Console",
        "Intended Audience :: Developers",
        "Topic :: Software Development :: Libraries",
        # Indique licencia usada (debe coincidir con el "license")
        "License :: OSI Approved :: GNU General Public License",
        # Indique versiones soportas, Python 2, Python 3 o ambos.
        "Programming Language :: Python",
        "Programming Language :: Python :: 2.7",
        "Operating System :: OS Independent",
    ],
    keywords="ejemplo instalador paquete tostadas_pipo",
    author="Leonardo J. Caballero G.",
    author_email="leonardocaballero@gmail.com",
    url="https://twitter/macagua",
    download_url="https://github.com/macagua/tostadas_pipo",
    license="GPL",
    platforms="Unix",
    packages=["tostadas_pipo", "tostadas_pipo/utilidades/"],
    include_package_data=True,
)
```

Entonces debe cree la siguiente estructura de directorios, ya hecha para seguir adelante:

```
distribucion/
├── LICENSE
├── MANIFEST.in
├── README.txt
├── setup.py
└── tostadas_pipo
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ├── __init__.py
    ├── principal.py
    └── utilidades
        ├── calculos.py
        ├── impuestos.py
        └── __init__.py
```

8.3.4 Construir dependencias

Para construir cualquier cosas requeridas para instalar el paquete, ejecutando el siguiente comando:

```
python ./setup.py -v build
running build
running build_py
creating build
creating build/lib.linux-x86_64-2.7
creating build/lib.linux-x86_64-2.7/tostadas_pipo
copying tostadas_pipo/__init__.py -> build/lib.linux-x86_64-2.7/tostadas_pipo
copying tostadas_pipo/principal.py -> build/lib.linux-x86_64-2.7/tostadas_pipo
creating build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades
copying tostadas_pipo/utilidades/__init__.py -> build/lib.linux-x86_64-2.7/
↪tostadas_pipo/utilidades
copying tostadas_pipo/utilidades/calculos.py -> build/lib.linux-x86_64-2.7/
↪tostadas_pipo/utilidades
copying tostadas_pipo/utilidades/impuestos.py -> build/lib.linux-x86_64-2.7/
↪tostadas_pipo/utilidades
not copying tostadas_pipo/principal.py (output up-to-date)
not copying tostadas_pipo/__init__.py (output up-to-date)
```

De esta forma al terminar la ejecución del comando previo debe tener creado un directorio llamado `build` e incluyendo el paquete `tostadas_pipo` construido con todo lo necesario para crear su distribución, como se muestra a continuación:

```
build/
└── lib.linux-x86_64-2.7
    └── tostadas_pipo
        ├── __init__.py
        ├── principal.py
        └── utilidades
            ├── calculos.py
            ├── impuestos.py
            └── __init__.py
```

De esta forma ya construyo el paquete `tostadas_pipo` y todas las cosas necesarias para crear su distribución de código fuente o binaria para su proyecto.

8.3.5 Crear paquete

Usted puede crear diversos tipos de formatos de instalación y distribución de sus paquetes Python, a continuación se describen los mas usados:

Distribución código fuente

Tanto el módulo *setuptools* (página 164) y *distutils* (página 164) le permiten crear una distribución de código fuente o source distribution (`sdist`) de su paquete en formatos como **tarball**, archivo **zip**, etc. Para crear una paquete `sdist`, ejecute el siguiente comando:

```
python ./setup.py -v sdist
running sdist
running egg_info
creating tostadas_pipo.egg-info
writing tostadas_pipo.egg-info/PKG-INFO
writing top-level names to tostadas_pipo.egg-info/top_level.txt
writing dependency_links to tostadas_pipo.egg-info/dependency_links.txt
writing entry points to tostadas_pipo.egg-info/entry_points.txt
writing manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
reading manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
warning: no previously-included files matching '*.pyc' found anywhere in␣
↪distribution
warning: no previously-included files matching '*.pyo' found anywhere in␣
↪distribution
warning: no previously-included files matching '*~' found anywhere in distribution
no previously-included directories found matching 'build'
no previously-included directories found matching 'dist'
writing manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
running check
creating tostadas_pipo-0.1
creating tostadas_pipo-0.1/tostadas_pipo
creating tostadas_pipo-0.1/tostadas_pipo.egg-info
creating tostadas_pipo-0.1/tostadas_pipo/utilidades
copying files to tostadas_pipo-0.1...
copying LICENSE -> tostadas_pipo-0.1
copying MANIFEST.in -> tostadas_pipo-0.1
copying README.txt -> tostadas_pipo-0.1
copying setup.py -> tostadas_pipo-0.1
copying tostadas_pipo/__init__.py -> tostadas_pipo-0.1/tostadas_pipo
copying tostadas_pipo/principal.py -> tostadas_pipo-0.1/tostadas_pipo
copying tostadas_pipo.egg-info/PKG-INFO -> tostadas_pipo-0.1/tostadas_pipo.egg-info
copying tostadas_pipo.egg-info/SOURCES.txt -> tostadas_pipo-0.1/tostadas_pipo.egg-
↪info
copying tostadas_pipo.egg-info/dependency_links.txt -> tostadas_pipo-0.1/tostadas_
↪pipo.egg-info
copying tostadas_pipo.egg-info/entry_points.txt -> tostadas_pipo-0.1/tostadas_pipo.
↪egg-info
copying tostadas_pipo.egg-info/top_level.txt -> tostadas_pipo-0.1/tostadas_pipo.
↪egg-info
copying tostadas_pipo/utilidades/__init__.py -> tostadas_pipo-0.1/tostadas_pipo/
↪utilidades
copying tostadas_pipo/utilidades/calculos.py -> tostadas_pipo-0.1/tostadas_pipo/
↪utilidades
copying tostadas_pipo/utilidades/impuestos.py -> tostadas_pipo-0.1/tostadas_pipo/
↪utilidades
not copying tostadas_pipo.egg-info/SOURCES.txt (output up-to-date)
Reading configuration from tostadas_pipo-0.1/setup.cfg
Adding new section [egg_info] to tostadas_pipo-0.1/setup.cfg
Setting egg_info.tag_build to '' in tostadas_pipo-0.1/setup.cfg
Setting egg_info.tag_date to 0 in tostadas_pipo-0.1/setup.cfg
Writing tostadas_pipo-0.1/setup.cfg
creating dist
Creating tar archive
removing 'tostadas_pipo-0.1' (and everything under it)
```

De esta forma al terminar la ejecución del comando previo debe tener creado un directorio llamado `dist` e incluyendo el paquete en formato de archivo tarball comprimido en *gztar*, como se muestra a continuación:

```
dist/
└── tostadas_pipo-0.1.tar.gz
```

Por defecto, tanto el módulo *setuptools* (página 164) y *distutils* (página 164) creá el paquete en formato de archivo tarball comprimido usando *gztar*.).

Usted puede cambiar el formato de paquete a crear de su distribución de código fuente de su paquete (en formato archivo **tarball**, archivo **zip**, etc.), ejecute el siguiente comando:

```
python ./setup.py sdist --formats=zip,gztar,bztar
running sdist
running egg_info
writing tostadas_pipo.egg-info/PKG-INFO
writing top-level names to tostadas_pipo.egg-info/top_level.txt
writing dependency_links to tostadas_pipo.egg-info/dependency_links.txt
writing entry points to tostadas_pipo.egg-info/entry_points.txt
reading manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
warning: no previously-included files matching '*.pyc' found anywhere in␣
↪distribution
warning: no previously-included files matching '*.pyo' found anywhere in␣
↪distribution
warning: no previously-included files matching '*~' found anywhere in distribution
no previously-included directories found matching 'build'
no previously-included directories found matching 'dist'
writing manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
running check
creating tostadas_pipo-0.1
creating tostadas_pipo-0.1/tostadas_pipo
creating tostadas_pipo-0.1/tostadas_pipo.egg-info
creating tostadas_pipo-0.1/tostadas_pipo/utilidades
copying files to tostadas_pipo-0.1...
copying LICENSE -> tostadas_pipo-0.1
copying MANIFEST.in -> tostadas_pipo-0.1
copying README.txt -> tostadas_pipo-0.1
copying setup.py -> tostadas_pipo-0.1
copying tostadas_pipo/__init__.py -> tostadas_pipo-0.1/tostadas_pipo
copying tostadas_pipo/principal.py -> tostadas_pipo-0.1/tostadas_pipo
copying tostadas_pipo.egg-info/PKG-INFO -> tostadas_pipo-0.1/tostadas_pipo.egg-info
copying tostadas_pipo.egg-info/SOURCES.txt -> tostadas_pipo-0.1/tostadas_pipo.egg-
↪info
copying tostadas_pipo.egg-info/dependency_links.txt -> tostadas_pipo-0.1/tostadas_
↪pipo.egg-info
copying tostadas_pipo.egg-info/entry_points.txt -> tostadas_pipo-0.1/tostadas_pipo.
↪egg-info
copying tostadas_pipo.egg-info/top_level.txt -> tostadas_pipo-0.1/tostadas_pipo.
↪egg-info
copying tostadas_pipo/utilidades/__init__.py -> tostadas_pipo-0.1/tostadas_pipo/
↪utilidades
copying tostadas_pipo/utilidades/calculos.py -> tostadas_pipo-0.1/tostadas_pipo/
↪utilidades
copying tostadas_pipo/utilidades/impuestos.py -> tostadas_pipo-0.1/tostadas_pipo/
↪utilidades
Writing tostadas_pipo-0.1/setup.cfg
creating 'dist/tostadas_pipo-0.1.zip' and adding 'tostadas_pipo-0.1' to it
adding 'tostadas_pipo-0.1/MANIFEST.in'
adding 'tostadas_pipo-0.1/setup.cfg'
adding 'tostadas_pipo-0.1/PKG-INFO'
adding 'tostadas_pipo-0.1/LICENSE'
adding 'tostadas_pipo-0.1/README.txt'
adding 'tostadas_pipo-0.1/setup.py'
adding 'tostadas_pipo-0.1/tostadas_pipo/principal.py'
adding 'tostadas_pipo-0.1/tostadas_pipo/__init__.py'
adding 'tostadas_pipo-0.1/tostadas_pipo/utilidades/impuestos.py'
adding 'tostadas_pipo-0.1/tostadas_pipo/utilidades/__init__.py'
adding 'tostadas_pipo-0.1/tostadas_pipo/utilidades/calculos.py'
```

(continué en la próxima página)

(proviene de la página anterior)

```
adding 'tostadas_pipo-0.1/tostadas_pipo.egg-info/dependency_links.txt'
adding 'tostadas_pipo-0.1/tostadas_pipo.egg-info/entry_points.txt'
adding 'tostadas_pipo-0.1/tostadas_pipo.egg-info/PKG-INFO'
adding 'tostadas_pipo-0.1/tostadas_pipo.egg-info/SOURCES.txt'
adding 'tostadas_pipo-0.1/tostadas_pipo.egg-info/top_level.txt'
Creating tar archive
Creating tar archive
removing 'tostadas_pipo-0.1' (and everything under it)
```

De esta forma al terminar la ejecución del comando previo debe tener creado un directorio llamado `dist` e incluyendo los tres paquetes en formatos de archivos tarball comprimido en *gzip/bzip2* y archivo comprimido en *zip*.

```
dist/
├── tostadas_pipo-0.1.tar.bz2
├── tostadas_pipo-0.1.tar.gz
└── tostadas_pipo-0.1.zip
```

De esta forma ya creo el(los) paquete(s) en diversos formato de distribución de código fuente para su proyecto.

Distribución binaria

El módulo *setuptools* (página 164) y *distutils* (página 164) le permiten crear una distribución binaria construida o built «binary» distribution (`bdist`) de su paquete en formato **egg**, **wheel**, **rpm**, etc. A continuación se describen los mas usados:

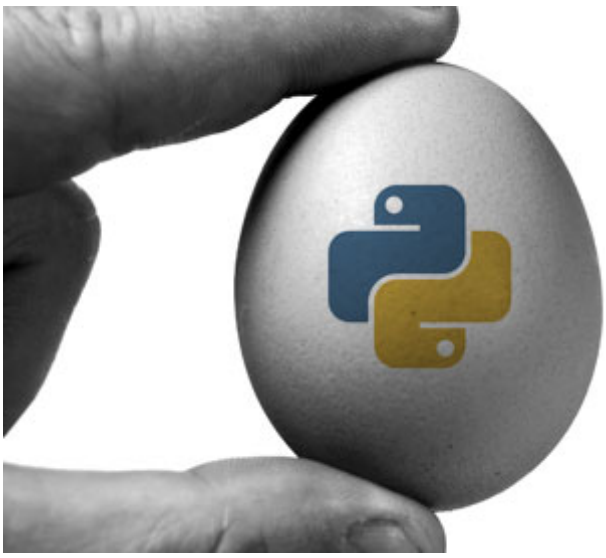

Figura 8.1: Distribución binaria Egg.

Egg

Para crear una distribución `bdist` de su paquete en formato `egg`, ejecute el siguiente comando:

```
python ./setup.py bdist_egg
running bdist_egg
running egg_info
writing tostadas_pipo.egg-info/PKG-INFO
writing top-level names to tostadas_pipo.egg-info/top_level.txt
writing dependency_links to tostadas_pipo.egg-info/dependency_links.txt
```

(continué en la próxima página)

(proviene de la página anterior)

```
writing entry points to tostadas_pipo.egg-info/entry_points.txt
reading manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
warning: no previously-included files matching '*.pyc' found anywhere in␣
↪distribution
warning: no previously-included files matching '*.pyo' found anywhere in␣
↪distribution
warning: no previously-included files matching '*~' found anywhere in distribution
no previously-included directories found matching 'build'
no previously-included directories found matching 'dist'
writing manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
running build_py
creating build/bdist.linux-x86_64
creating build/bdist.linux-x86_64/egg
creating build/bdist.linux-x86_64/egg/tostadas_pipo
copying build/lib.linux-x86_64-2.7/tostadas_pipo/principal.py -> build/bdist.linux-
↪x86_64/egg/tostadas_pipo
copying build/lib.linux-x86_64-2.7/tostadas_pipo/__init__.py -> build/bdist.linux-
↪x86_64/egg/tostadas_pipo
creating build/bdist.linux-x86_64/egg/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/impuestos.py -> build/
↪bdist.linux-x86_64/egg/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/__init__.py -> build/
↪bdist.linux-x86_64/egg/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/calculos.py -> build/
↪bdist.linux-x86_64/egg/tostadas_pipo/utilidades
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/principal.py to␣
↪principal.pyc
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/__init__.py to __init__.
↪pyc
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/utilidades/impuestos.py␣
↪to impuestos.pyc
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/utilidades/__init__.py␣
↪to __init__.pyc
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/utilidades/calculos.py␣
↪to calculos.pyc
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying tostadas_pipo.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
copying tostadas_pipo.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying tostadas_pipo.egg-info/dependency_links.txt -> build/bdist.linux-x86_64/
↪egg/EGG-INFO
copying tostadas_pipo.egg-info/entry_points.txt -> build/bdist.linux-x86_64/egg/
↪EGG-INFO
copying tostadas_pipo.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/EGG-
↪INFO
zip_safe flag not set; analyzing archive contents...
creating 'dist/tostadas_pipo-0.1-py2.7.egg' and adding 'build/bdist.linux-x86_64/
↪egg' to it
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
```

De esta forma al terminar la ejecución del comando previo debe tener creado un directorio llamado `dist` e incluyendo la distribución `bdist` en formato *egg*.

```
dist/
└── tostadas_pipo-0.1-py2.7.egg
```

De esta forma ya creo la distribución `bdist` del paquete en formato *egg* para su proyecto.

Wheel

Para crear una distribución `bdist` de su paquete en formato **wheel**, ejecute el siguiente comando:

```
python ./setup.py bdist_wheel
running bdist_wheel
running build
running build_py
installing to build/bdist.linux-x86_64/wheel
running install
running install_lib
creating build/bdist.linux-x86_64/wheel
creating build/bdist.linux-x86_64/wheel/tostadas_pipo
copying build/lib.linux-x86_64-2.7/tostadas_pipo/principal.py -> build/bdist.linux-
↪x86_64/wheel/tostadas_pipo
copying build/lib.linux-x86_64-2.7/tostadas_pipo/__init__.py -> build/bdist.linux-
↪x86_64/wheel/tostadas_pipo
creating build/bdist.linux-x86_64/wheel/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/impuestos.py -> build/
↪bdist.linux-x86_64/wheel/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/__init__.py -> build/
↪bdist.linux-x86_64/wheel/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/calculos.py -> build/
↪bdist.linux-x86_64/wheel/tostadas_pipo/utilidades
running install_egg_info
running egg_info
writing tostadas_pipo.egg-info/PKG-INFO
writing top-level names to tostadas_pipo.egg-info/top_level.txt
writing dependency_links to tostadas_pipo.egg-info/dependency_links.txt
writing entry points to tostadas_pipo.egg-info/entry_points.txt
reading manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
warning: no previously-included files matching '*.pyc' found anywhere in␣
↪distribution
warning: no previously-included files matching '*.pyo' found anywhere in␣
↪distribution
warning: no previously-included files matching '*~' found anywhere in distribution
no previously-included directories found matching 'build'
no previously-included directories found matching 'dist'
writing manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
Copying tostadas_pipo.egg-info to build/bdist.linux-x86_64/wheel/tostadas_pipo-0.1-
↪py2.7.egg-info
running install_scripts
adding license file "LICENSE" (matched pattern "LICEN[CS]E*")
creating build/bdist.linux-x86_64/wheel/tostadas_pipo-0.1.dist-info/WHEEL
creating 'dist/tostadas_pipo-0.1-py2-none-any.whl' and adding 'build/bdist.linux-
↪x86_64/wheel' to it
adding 'tostadas_pipo/__init__.py'
adding 'tostadas_pipo/principal.py'
adding 'tostadas_pipo/utilidades/__init__.py'
adding 'tostadas_pipo/utilidades/calculos.py'
adding 'tostadas_pipo/utilidades/impuestos.py'
adding 'tostadas_pipo-0.1.dist-info/LICENSE'
adding 'tostadas_pipo-0.1.dist-info/METADATA'
adding 'tostadas_pipo-0.1.dist-info/WHEEL'
adding 'tostadas_pipo-0.1.dist-info/entry_points.txt'
adding 'tostadas_pipo-0.1.dist-info/top_level.txt'
adding 'tostadas_pipo-0.1.dist-info/RECORD'
removing build/bdist.linux-x86_64/wheel
```

De esta forma al terminar la ejecución del comando previo debe tener creado un directorio llamado `dist` e incluyendo la distribución `bdist` en formato *whl*.

```
dist/
├── tostadas_pipo-0.1-py2.7.egg
└── tostadas_pipo-0.1-py2-none-any.whl
```

De esta forma ya creo la distribución `bdist` del paquete en formato *whl* para su proyecto.

8.3.6 Instalar paquete

Para instalar el paquete de su proyecto, hay dos formas de instalación disponibles a continuación:

Instalar distribución código fuente

Para instalar una distribución código fuente de su paquete previamente creado, se realizar usando la herramienta `pip`, ejecutando el siguiente comando:

```
pip install --user dist/tostadas_pipo-0.1.tar.gz
```

Si al ejecutar el comando anterior muestra el mensaje:

```
pip
bash: pip: no se encontró la orden
```

Esto es debido a que no tiene instalado dicha herramienta, así que debe ejecutar el siguiente comando:

```
sudo apt-get install -y python-pip
```

De nuevo vuelva a ejecutar en su consola de comando el comando:

```
pip install --user dist/tostadas_pipo-0.1.tar.gz
Processing ./dist/tostadas_pipo-0.1.tar.gz
Building wheels for collected packages: tostadas-pipo
  Running setup.py bdist_wheel for tostadas-pipo ... done
  Stored in directory: /home/leonardo/.cache/pip/wheels/fd/f9/75/
↪a6965566a3c5a8bff507d7daa30760caca0a7525a3de61eac2
Successfully built tostadas-pipo
Installing collected packages: tostadas-pipo
Successfully installed tostadas-pipo-0.1
```

De esta forma tiene instalado una distribución código fuente en formato **tarball** de su paquete en el interprete Python usando la herramienta `pip`.

Instalar distribución binaria

Para instalar una distribución binaria de su paquete previamente creado, se realizar usando la herramienta `pip`, ejecutando el siguiente comando:

```
pip install --user ./dist/tostadas_pipo-0.1-py2-none-any.whl
Processing ./dist/tostadas_pipo-0.1-py2-none-any.whl
Installing collected packages: tostadas-pipo
Successfully installed tostadas-pipo-0.1
```

De esta forma tiene instalado una distribución binaria en formato **wheel** de su paquete en el interprete Python usando la herramienta `pip`.

Nota: pip[67], es una herramienta para instalación y administración de paquetes Python.

---

[67] https://pip.readthedocs.io/

Instalar de código de proyecto

Para instalar el paquete desde el código de proyecto, ejecute el siguiente comando:

```
python ./setup.py -v install --user
running install
running bdist_egg
running egg_info
writing tostadas_pipo.egg-info/PKG-INFO
writing top-level names to tostadas_pipo.egg-info/top_level.txt
writing dependency_links to tostadas_pipo.egg-info/dependency_links.txt
reading manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
reading manifest template 'MANIFEST.in'
warning: no previously-included files matching '*.pyo' found anywhere in␣
↪distribution
warning: no previously-included files matching '*~' found anywhere in distribution
no previously-included directories found matching 'build'
no previously-included directories found matching 'dist'
writing manifest file 'tostadas_pipo.egg-info/SOURCES.txt'
installing library code to build/bdist.linux-x86_64/egg
running install_lib
running build_py
not copying tostadas_pipo/principal.py (output up-to-date)
not copying tostadas_pipo/__init__.py (output up-to-date)
not copying tostadas_pipo/utilidades/impuestos.py (output up-to-date)
not copying tostadas_pipo/utilidades/__init__.py (output up-to-date)
not copying tostadas_pipo/utilidades/calculos.py (output up-to-date)
not copying tostadas_pipo/utilidades/__init__.py (output up-to-date)
not copying tostadas_pipo/utilidades/calculos.py (output up-to-date)
not copying tostadas_pipo/utilidades/impuestos.py (output up-to-date)
creating build/bdist.linux-x86_64
creating build/bdist.linux-x86_64/egg
creating build/bdist.linux-x86_64/egg/tostadas_pipo
copying build/lib.linux-x86_64-2.7/tostadas_pipo/principal.py -> build/bdist.linux-
↪x86_64/egg/tostadas_pipo
copying build/lib.linux-x86_64-2.7/tostadas_pipo/__init__.py -> build/bdist.linux-
↪x86_64/egg/tostadas_pipo
creating build/bdist.linux-x86_64/egg/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/impuestos.py -> build/
↪bdist.linux-x86_64/egg/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/__init__.py -> build/
↪bdist.linux-x86_64/egg/tostadas_pipo/utilidades
copying build/lib.linux-x86_64-2.7/tostadas_pipo/utilidades/calculos.py -> build/
↪bdist.linux-x86_64/egg/tostadas_pipo/utilidades
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/principal.py to␣
↪principal.pyc
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/__init__.py to __init__.
↪pyc
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/utilidades/impuestos.py␣
↪to impuestos.pyc
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/utilidades/__init__.py␣
↪to __init__.pyc
byte-compiling build/bdist.linux-x86_64/egg/tostadas_pipo/utilidades/calculos.py␣
↪to calculos.pyc
creating build/bdist.linux-x86_64/egg/EGG-INFO
copying tostadas_pipo.egg-info/PKG-INFO -> build/bdist.linux-x86_64/egg/EGG-INFO
copying tostadas_pipo.egg-info/SOURCES.txt -> build/bdist.linux-x86_64/egg/EGG-INFO
copying tostadas_pipo.egg-info/dependency_links.txt -> build/bdist.linux-x86_64/
↪egg/EGG-INFO
```

(continué en la próxima página)

(proviene de la página anterior)

```
copying tostadas_pipo.egg-info/top_level.txt -> build/bdist.linux-x86_64/egg/EGG-
↪INFO
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist/tostadas_pipo-0.1-py2.7.egg' and adding 'build/bdist.linux-x86_64/
↪egg' to it
adding 'tostadas_pipo/principal.py'
adding 'tostadas_pipo/principal.pyc'
adding 'tostadas_pipo/__init__.pyc'
adding 'tostadas_pipo/__init__.py'
adding 'tostadas_pipo/utilidades/impuestos.pyc'
adding 'tostadas_pipo/utilidades/calculos.pyc'
adding 'tostadas_pipo/utilidades/impuestos.py'
adding 'tostadas_pipo/utilidades/__init__.pyc'
adding 'tostadas_pipo/utilidades/__init__.py'
adding 'tostadas_pipo/utilidades/calculos.py'
adding 'EGG-INFO/zip-safe'
adding 'EGG-INFO/dependency_links.txt'
adding 'EGG-INFO/PKG-INFO'
adding 'EGG-INFO/SOURCES.txt'
adding 'EGG-INFO/top_level.txt'
removing 'build/bdist.linux-x86_64/egg' (and everything under it)
Processing tostadas_pipo-0.1-py2.7.egg
Copying tostadas_pipo-0.1-py2.7.egg to /home/leonardo/.local/lib/python2.7/site-
↪packages
Adding tostadas-pipo 0.1 to easy-install.pth file
Saving /home/leonardo/.local/lib/python2.7/site-packages/easy-install.pth

Installed /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo-0.1-py2.
↪7.egg
Processing dependencies for tostadas-pipo==0.1
Finished processing dependencies for tostadas-pipo==0.1
```

De esta forma tiene instalado su paquete en su interprete Python usando el comando `install` disponible con el script `setup.py`.

Advertencia: Al instalar el paquete usando el parámetro `--user` el paquete es instalado en el directorio `$HOME/.local/lib/python2.7/site-packages/`.

Comprobar la instalación

Usted puede comprobar luego de realizar la instalación de la distribución de código fuente o binaria de su paquete, ejecute el siguiente comando:

```
pip list --user --format=freeze | grep "tostadas"
tostadas-pipo==0.1
```

De esta forma la herramienta de gestión de paquete indica que el `tostadas-pipo` en su versión **0.1** esta instalado en su interprete Python.

8.3.7 Usar paquete

Usar el paquete `tostadas_pipo-0.1`, recuerde que debe usarlo como una librería, entonces puede probar el correcto funcionamiento del paquete, importando este, ejecutando el siguiente comando:

```
python -c 'from tostadas_pipo.utilidades.impuestos import impuesto_iva14; print
↪"Función importada " + impuesto_iva14.__doc__[1:36] + "."'
Función importada Calcula el impuesto del IVA de 14 %.
```

El comando previo muestra la *docstring* (página 50) de la función importada `impuesto_iva14` sino muestra ningún mensaje de error, el paquete `tostadas_pipo-0.1` se instalo correctamente.

8.3.8 Eliminar paquete

Para eliminar paquete usando la herramienta `pip`, ejecute el siguiente comando:

```
pip uninstall tostadas_pipo
Uninstalling tostadas-pipo-0.1:
  /home/leonardo/.local/bin/tostadas_pipo
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo-0.1.dist-info/
↪DESCRIPTION.rst
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo-0.1.dist-info/
↪INSTALLER
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo-0.1.dist-info/
↪METADATA
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo-0.1.dist-info/
↪RECORD
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo-0.1.dist-info/
↪WHEEL
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo-0.1.dist-info/
↪metadata.json
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo-0.1.dist-info/
↪top_level.txt
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/__init__.py
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/__init__.pyc
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/principal.py
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/principal.pyc
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/utilidades/__
↪init__.py
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/utilidades/__
↪init__.pyc
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/utilidades/
↪calculos.py
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/utilidades/
↪calculos.pyc
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/utilidades/
↪impuestos.py
  /home/leonardo/.local/lib/python2.7/site-packages/tostadas_pipo/utilidades/
↪impuestos.pyc
Proceed (y/n)? y
  Successfully uninstalled tostadas-pipo-0.1
```

`pip` esta habilitado a desinstalar la mayoría de paquetes instalados. Las excepciones conocidas son:

- Los paquetes basado en solamente en el módulo *distutils* (página 164) los cuales fueron instalados sin la herramienta `pip` usando el comando `python setup.py install` desde el *código del paquete* (página 174).

  Instalándolo de esta forma, al momento de desintalarlo usando el comando `pip uninstall tostadas_pipo` este comando removerá solo la metadata, no detrás dejando de la instalación metadata para determinar que archivos fueron instalados.

  Entonces para solventar este problema tiene que ir manualmente al directorio `site-packages` a eliminar manualmente el paquete que instalo.

> **Advertencia:** Esta entrando a la cueva de los Dragones!!!

- Los scripts wrappers instalados ejecutando el comando `python setup.py develop`.

De esta forma ya tiene eliminado su paquete de forma manual de su sistema.

8.3.9 Ayuda integrada

Usted puede consultar toda la ayuda comandos disponibles del módulo *setuptools* (página 164) y *distutils* (página 164), ejecute el comando siguiente:

```
python ./setup.py --help-commands
Standard commands:
  build             build everything needed to install
  build_py          "build" pure Python modules (copy to build directory)
  build_ext         build C/C++ extensions (compile/link to build directory)
  build_clib        build C/C++ libraries used by Python extensions
  build_scripts     "build" scripts (copy and fixup #! line)
  clean             clean up temporary files from 'build' command
  install           install everything from build directory
  install_lib       install all Python modules (extensions and pure Python)
  install_headers   install C/C++ header files
  install_scripts   install scripts (Python or otherwise)
  install_data      install data files
  sdist             create a source distribution (tarball, zip file, etc.)
  register          register the distribution with the Python package index
  bdist             create a built (binary) distribution
  bdist_dumb        create a "dumb" built distribution
  bdist_rpm         create an RPM distribution
  bdist_wininst     create an executable installer for MS Windows
  upload            upload binary package to PyPI
  check             perform some checks on the package

Extra commands:
  saveopts          save supplied options to setup.cfg or other config file
  testr             Run unit tests using testr
  compile_catalog   compile message catalogs to binary MO files
  develop           install package in 'development mode'
  upload_docs       Upload documentation to PyPI
  extract_messages  extract localizable strings from the project code
  init_catalog      create a new catalog based on a POT file
  test              run unit tests after in-place build
  update_catalog    update message catalogs from a POT file
  setopt            set an option in setup.cfg or another config file
  install_egg_info  Install an .egg-info directory for the package
  rotate            delete older distributions, keeping N newest files
  bdist_wheel       create a wheel distribution
  egg_info          create a distribution's .egg-info directory
  alias             define a shortcut to invoke one or more commands
  easy_install      Find/get/install Python packages
  bdist_egg         create an "egg" distribution

usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help
```

Para consultar toda la ayuda del módulo *setuptools* (página 164) y *distutils* (página 164), ejecute el comando siguiente:

```
python setup.py --help
```

Importante: Usted puede descargar el código usado en esta sección, haciendo clic en el siguiente enlace: `distribucion.zip`.

Truco: Para poder definir un instalador y construirlo para así poder hacer que su proyecto se pueda distribuir de forma más fácil debe crear la *estructura de proyecto* (página 164) usando el código descomprimido del archivo `distribucion.zip`, siga los pasos para construir los archivos, crear el instalador y probar su instalación.

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

8.4 Scaffolding en proyectos Python

Sobre este artículo

Autor(es) Leonardo J. Caballero G.

Correo(s) leonardoc@plone.org

Compatible con Python 2.4 o versiones superiores

Fecha 11 de Enero de 2021

La estructura del *paquete Egg* Python es poco compleja. Por lo cual para empezar con su primer proyecto y diversos módulos, puede usar el concepto **Scaffolding** para crear un esqueleto de código usando las plantillas adecuadas para *paquetes Python*.

Este concepto *scaffolding*, es muy útil para del arranque de su desarrollo, ofreciendo una serie de colecciones de plantillas *esqueletos* que permiten iniciar rápidamente proyectos, existente diversos *esqueletos* orientados a tipos de desarrollos específicos.

8.4.1 ¿Qué es PasteScript?

Es una herramienta de linea de comando basada en plugins que le permiten crear estructuras de paquetes de proyectos Python además sirve aplicaciones web, con configuraciones basadas en paste.deploy[68].

Instalación

Dentro de su entorno virtual[69] activado debe instalar el paquete PasteScript[70], ejecutando el siguiente comando:

```
(python)$ pip install PasteScript
```

[68] https://pypi.org/project/PasteDeploy
[69] https://plone-spanish-docs.readthedocs.io/es/latest/python/creacion_entornos_virtuales.html
[70] https://pypi.org/project/PasteScript

Nota: No olvidar que estos paquetes han sido instalados con el entorno virtual que previamente usted activo, eso quiere decir que los paquetes previamente instalados con easy_install[71] están instalados en el directorio `~/virtualenv/python/lib/python2.x/site-packages/` en ves del directorio de su versión de Python de sistema `/usr/lib/python2.x/site-packages/`

Al finalizar la instalación podrá opcionalmente consultar cuales plantillas tiene disponible para usa, ejecutando el siguiente comando:

```
(python)$ paster create --list-templates
  Available templates:
    basic_package:       A basic setuptools-enabled package
    paste_deploy:        A web application deployed through paste.deploy
```

Usted puede usar el comando **`paster`** para crear paquetes Python.

```
(python)$ paster create -t basic_package mipaquetepython

  Selected and implied templates:

    PasteScript#basic_package  A basic setuptools-enabled package

  Variables:
    egg:      mipaquetepython
    package:  mipaquetepython
    project:  mipaquetepython
  Enter version (Version (like 0.1)) ['']: 0.1
  Enter description (One-line description of the package) ['']: Mi Paquete Básico
  Enter long_description (Multi-line description (in reST)) ['']: Mi Paquete␣
↪Básico para demostrar el uso de PasteScript
  Enter keywords (Space-separated keywords/tags) ['']: PasteScript Basic Package␣
↪Demo
  Enter author (Author name) ['']: Pedro Picapiedra
  Enter author_email (Author email) ['']: pedro@acme.com
  Enter url (URL of homepage) ['']: https://github.com/pyve/mipaquetepython
  Enter license_name (License name) ['']: GPL
  Enter zip_safe (True/False: if the package can be distributed as a .zip file)␣
↪[False]:
  Creating template basic_package
  Creating directory ./mipaquetepython
    Recursing into +package+
      Creating ./mipaquetepython/mipaquetepython/
      Copying __init__.py to
      ./mipaquetepython/mipaquetepython/__init__.py
    Copying setup.cfg to ./mipaquetepython/setup.cfg
    Copying setup.py_tmpl to ./mipaquetepython/setup.py
  Running /home/macagua/virtualenv/python/bin/python setup.py egg_info
```

Usted puede verificar el paquete previamente creado y observará como este paquete básico ha habilitado el Setuptools[72].

```
(python)$ tree mipaquetepython/
  mipaquetepython/
  |-- mipaquetepython
  |   `-- __init__.py
  |-- mipaquetepython.egg-info
  |   |-- PKG-INFO
  |   |-- SOURCES.txt
  |   |-- dependency_links.txt
```

(continué en la próxima página)

[71] https://plone-spanish-docs.readthedocs.io/es/latest/python/setuptools.html#que-es-easyinstall
[72] https://plone-spanish-docs.readthedocs.io/es/latest/python/setuptools.html

(proviene de la página anterior)

```
|   |-- entry_points.txt
|   |-- not-zip-safe
|   `-- top_level.txt
|-- setup.cfg
`-- setup.py
```

Para instalar este paquete ejecute el siguiente comando:

```
(python)$ cd mipaquetepython/mipaquetepython/
(python)$ vim app.py
```

Escriba un simple código que solicita un valor y luego lo muestra:

```
var = raw_input("Introduzca alguna frase: ")
print "Usted introdujo: ", var
```

Guarde los cambios en el archivo `app.py`, luego importe su aplicación `app.py` en el archivo `__init__.py` con el siguiente código fuente:

```
from mipaquetepython import app
```

Para comprobar su instalación ejecute el siguiente comando:

```
(python)$ python
```

Y realice una importación del paquete `mipaquetepython` ejecutando el siguiente comando:

```
>>> import mipaquetepython
Introduzca alguna frase: Esta cadena
Usted introdujo:  Esta cadena
>>> exit()
```

De esta forma tienes creado un *paquete Egg* Python.

8.4.2 Esqueletos en diversos proyectos Python

A continuación se muestran algunos esqueletos útiles:

- Esqueletos de proyectos Zope/Plone[73].
- Esqueletos de proyectos Odoo (Antiguo OpenERP)[74].

  Nota: Odoo[75], es un sistema ERP y CRM programado con Python, de propósito general.

- **Esqueletos de proyectos Django**:

  Nota: Django[76], es un Framework Web Python, de propósito general.

  - django-project-templates[77], plantillas Paster para crear proyectos Django.
  - fez.djangoskel[78], es una colección de plantillas Paster para crear aplicaciones Django como *paquetes Egg*.

[73] https://plone-spanish-docs.readthedocs.io/es/latest/python/skel_proyectos_plone.html
[74] https://plone-spanish-docs.readthedocs.io/es/latest/python/skel_proyectos_openerp.html
[75] https://www.odoo.com/
[76] https://www.djangoproject.com/
[77] https://pypi.org/project/django-project-templates
[78] https://pypi.org/project/fez.djangoskel

- django-harness[79], es una aplicación destinada a simplificar las tareas típicas relacionadas con la creación de un sitio web hechos con Django, el mantenimiento de varias instalaciones (local, producción, etc) y cuidando su instalación global y su estructura de «esqueleto» actualizado del sitio de manera fácil.
- lfc-skel[80], Provee una plantilla para crear una aplicación django-lfc[81] CMS.

- **Esqueletos de proyectos Pylons**:

Nota: Pylons[82], es un Framework Web Python, de propósito general.

- Pylons[83], al instalarse usando la utilidad easy_install[84] instala dos plantillas de proyectos Pylons.
- PylonsTemplates[85], le ofrece plantillas adicionales `paster` para aplicaciones Pylons, incluyendo implementación de `repoze.what`.
- BlastOff[86], Una plantilla de aplicación Pylons[87] que proporciona un esqueleto de entorno de trabajo configurado con `SQLAlchemy`, `mako`, `repoze.who`, `ToscaWidgets`, `TurboMail`, `WebFlash` y (opcionalmente) `SchemaBot`. La aplicación generada esta previamente configurada con autenticación, inicio de sesión y formularios de registro, y (opcionalmente) confirmación de correo electrónico. `BlastOff` ayudar a acelerar el desarrollo de aplicaciones en Pylons por que genera un proyecto con una serie de dependencias configuraciones previamente.

- **Esqueletos de proyectos CherryPy**:

Nota: CherryPy[88], es un MicroFramework Web Python, de propósito general.

- CherryPaste[89], Usar CherryPy dentro Paste.

- **Esqueletos de proyectos Trac**:

Nota: Trac[90], es un sistema de gestión de proyectos de desarrollos de software.

- TracLegosScript[91], TracLegos es un software diseñado para ofrecer plantillas para proyectos Trac y asiste con la creación de proyecto trac.
- trac_project[92], Plantilla de proyecto Trac de software de código abierto.

8.4.3 Recomendaciones

Si desea trabajar con algún proyecto de desarrollo basado en esqueletos o plantillas `paster` y Buildout simplemente seleccione cual esqueleto va a utilizar para su desarrollo y proceso a instalarlo con easy_install[93] o PIP[94] (como se explico anteriormente) y siga sus respectivas instrucciones para lograr con éxito la tarea deseada.

[79] https://pypi.org/project/django-harness
[80] https://pypi.org/project/lfc-skel/
[81] https://pypi.org/project/django-lfc
[82] https://pypi.org/project/Pylons/
[83] https://pypi.org/project/Pylons/
[84] https://plone-spanish-docs.readthedocs.io/es/latest/python/setuptools.html#que-es-easyinstall
[85] https://pypi.org/project/PylonsTemplates/
[86] https://pypi.org/project/BlastOff/
[87] https://pypi.org/project/Pylons/
[88] https://pypi.org/project/CherryPy
[89] https://pypi.org/project/CherryPaste
[90] https://pypi.org/project/Trac
[91] https://trac-hacks.org/wiki/TracLegosScript
[92] https://trac-hacks.org/browser/traclegosscript/anyrelease/example/oss
[93] https://plone-spanish-docs.readthedocs.io/es/latest/python/setuptools.html#que-es-easyinstall
[94] https://plone-spanish-docs.readthedocs.io/es/latest/python/distribute_pip.html

8.4.4 Referencias

- Gestión de proyectos con Buildout, instalando Zope/Plone con este mecanismo[95] desde la comunidad de Plone Venezuela.

[95] https://coactivate.org/projects/ploneve/gestion-de-proyectos-con-buildout

CAPÍTULO 9

Manejos de errores y orientación a objetos

En Python usted puede manejar de los errores de sus aplicaciones, además de poder escribir sus aplicaciones usando el paradigma de la *orientación a objetos* (página 194).

En esta lección se describen el manejo de errores y la programación orientada a objetos con el lenguaje Python, mostrando ejemplos prácticos y útiles. A continuación el temario de esta lección:

9.1 Errores y excepciones

Hasta ahora los mensajes de error no habían sido más que mencionados, pero si probaste los ejemplos probablemente hayas visto algunos. Hay (al menos) dos tipos diferentes de errores: *errores de sintaxis* y *excepciones*.

9.1.1 Errores de sintaxis

Los errores de sintaxis, también conocidos como errores de interpretación, son quizás el tipo de queja más común que tenés cuando todavía estás aprendiendo Python:

```
>>> while True print 'Hola Mundo'
Traceback (most recent call last):
...
    while True print 'Hola Mundo'
                   ^
SyntaxError: invalid syntax
```

El intérprete repite la línea culpable y muestra una pequeña "flecha" que apunta al primer lugar donde se detectó el error. Este es causado por (o al menos detectado en) el símbolo que *precede* a la flecha: en el ejemplo, el error se detecta en la sentencia `print`, ya que faltan dos puntos (`':'`) antes del mismo. Se muestran el nombre del archivo y el número de línea para que sepas dónde mirar en caso de que la entrada venga de un programa.

9.1.2 Excepciones

Incluso si la sentencia o expresión es sintácticamente correcta, puede generar un error cuando se intenta ejecutarla. Los errores detectados durante la ejecución se llaman *excepciones*, y no son incondicionalmente fatales: pronto aprenderás cómo manejarlos en los programas en Python. Sin embargo, la mayoría de las excepciones no son manejadas por los programas, y resultan en mensajes de error como los mostrados aquí:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

La última línea de los mensajes de error indica qué sucedió. Las excepciones vienen de distintos tipos, y el tipo se imprime como parte del mensaje: los tipos en el ejemplo son: *ZeroDivisionError* (página 193), *NameError* (página 192) y *TypeError* (página 193).

La cadena mostrada como tipo de la excepción es el nombre de la excepción predefinida que ocurrió. Esto es verdad para todas las excepciones predefinidas del intérprete, pero no necesita ser verdad para excepciones definidas por el usuario (aunque es una convención útil). Los nombres de las excepciones estándar son identificadores incorporados al intérprete (no son palabras clave reservadas).

El resto de la línea provee un detalle basado en el tipo de la excepción y qué la causó.

La parte anterior del mensaje de error muestra el contexto donde la excepción sucedió, en la forma de un *trazado del error* listando líneas fuente; sin embargo, no mostrará líneas leídas desde la entrada estándar.

Excepciones integradas (página 190), es una lista las excepciones predefinidas y sus significados.

9.1.3 Manejando excepciones

Es posible escribir programas que manejen determinadas excepciones. Mirá el siguiente ejemplo, que le pide al usuario una entrada hasta que ingrese un entero válido, pero permite al usuario interrumpir el programa (usando `Control-C` o lo que sea que el sistema operativo soporte); notá que una interrupción generada por el usuario se señaliza generando la excepción *KeyboardInterrupt* (página 192).

```
>>> while True:
...     try:
...         x = int(raw_input(u"Por favor ingrese un número: "))
...         break
...     except ValueError:
...         print u"Oops!  No era válido. Intente nuevamente..."
...
```

La sentencia `try` funciona de la siguiente manera:

- Primero, se ejecuta el *bloque try* (el código entre las sentencias `try` y `except`).
- Si no ocurre ninguna excepción, el *bloque except* se saltea y termina la ejecución de la sentencia `try`.
- Si ocurre una excepción durante la ejecución del *bloque try*, el resto del bloque se saltea. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada `except`, se ejecuta el *bloque except*, y la ejecución continúa luego de la sentencia `try`.
- Si ocurre una excepción que no coincide con la excepción nombrada en el `except`, esta se pasa a declaraciones `try` de más afuera; si no se encuentra nada que la maneje, es una *excepción no manejada*, y la ejecución se frena con un mensaje como los mostrados arriba.

Una sentencia `try` puede tener más de un `except`, para especificar manejadores para distintas excepciones. A lo sumo un manejador será ejecutado. Sólo se manejan excepciones que ocurren en el correspondiente `try`, no en otros manejadores del mismo `try`. Un `except` puede nombrar múltiples excepciones usando paréntesis, por ejemplo:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

El último `except` puede omitir nombrar qué excepción captura, para servir como comodín. Usá esto con extremo cuidado, ya que de esta manera es fácil ocultar un error real de programación. También puede usarse para mostrar un mensaje de error y luego re-generar la excepción (permitiéndole al que llama, manejar también la excepción):

```
import sys

try:
    f = open('numeros.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "Error E/S ({0}): {1}".format(errno, strerror)
except ValueError:
    print "No pude convertir el dato a un entero."
except:
    print "Error inesperado:", sys.exc_info()[0]
    raise
```

Las declaraciones `try ... except` tienen un *bloque else* opcional, el cual, cuando está presente, debe seguir a los `except`. Es útil para aquel código que debe ejecutarse si el *bloque try* no genera una excepción. Por ejemplo:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'no pude abrir', arg
    else:
        print arg, 'tiene', len(f.readlines()), 'lineas'
        f.close()
```

El uso de `else` es mejor que agregar código adicional en el `try` porque evita capturar accidentalmente una excepción que no fue generada por el código que está protegido por la sentencia `try ... except`.

Cuando ocurre una excepción, puede tener un valor asociado, también conocido como el *argumento* de la excepción. La presencia y el tipo de argumento depende del tipo de excepción.

El `except` puede especificar una variable luego del nombre (o tupla) de excepción(es). La variable se vincula a una instancia de excepción con los argumentos almacenados en `instance.args`. Por conveniencia, la instancia de excepción define `__str__()` para que se pueda mostrar los argumentos directamente, sin necesidad de hacer referencia a `.args`.

Uno también puede instanciar una excepción antes de generarla, y agregarle cualquier atributo que se desee:

```
>>> try:
...     raise Exception('carne', 'huevos')
... except Exception as inst:
...     print type(inst)     # la instancia de excepción
...     print inst.args      # argumentos guardados en .args
...     print inst           # __str__ permite imprimir args directamente
...     x, y = inst          # __getitem__ permite usar args directamente
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('carne', 'huevos')
('carne', 'huevos')
x = carne
y = huevos
```

Si una excepción tiene un argumento, este se imprime como la última parte (el "detalle") del mensaje para las excepciones que no están manejadas.

Los manejadores de excepciones no manejan solamente las excepciones que ocurren en el *bloque try*, también manejan las excepciones que ocurren dentro de las funciones que se llaman (inclusive indirectamente) dentro del *bloque try*. Por ejemplo:

```
>>> def esto_falla():
...     x = 1/0
...
>>> try:
...     esto_falla()
... except ZeroDivisionError as detail:
...     print 'Manejando error en tiempo de ejecución:', detail
...
Manejando error en tiempo de ejecución: integer division or modulo by zero
```

9.1.4 Levantando excepciones

La sentencia `raise` permite al programador forzar a que ocurra una excepción específica. Por ejemplo:

```
>>> raise NameError('Hola')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: Hola
```

El único argumento a `raise` indica la excepción a generarse. Tiene que ser o una instancia de excepción, o una clase de excepción (una clase que hereda de *Exception* (página 190)).

Si necesitás determinar cuando una excepción fue lanzada pero no querés manejarla, una forma simplificada de la sentencia `raise` te permite relanzarla:

```
>>> try:
...     raise NameError('Hola')
... except NameError:
...     print u'Ha sucedido una excepción!'
...     raise
...
Ha sucedido una excepción!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: Hola
```

9.1.5 Sentencia assert

La sentencia `assert` es una vía conveniente para insertar afirmaciones de depuración dentro de un programa:

La forma simple, «assert expression», es equivalente a:

```
if __debug__:
    if not expression: raise AssertionError
```

La forma extendida, «assert expression1, expression2», es equivalente a:

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

Estas equivalencias suponen que `__debug__` y la excepción «*AssertionError* (página 191)» se refieren a las variables incorporadas con esos nombres. En la corriente implementación, la variable incorporada `__debug__` es `True` en circunstancias normales, `False` cuando se solicita la optimización (opción del línea de comando

`-O`). El generador de código actual no emite ningún código para una sentencia `assert` cuando se solicita la optimización en tiempo de compilación. Nota que no es necesario incluir el código fuente de la expresión que falló en el mensaje de error; se mostrará como parte del *stack trace*.

Asignaciones a `__debug__` son ilegales. El valor para la variable integrada es determinada cuando el interprete inicia.

9.1.6 Excepciones definidas por el usuario

Los programas pueden nombrar sus propias excepciones creando una nueva clase excepción (mirá el apartado de *Clases* (página 194) para más información sobre las clases de Python). Las excepciones, típicamente, deberán derivar de la clase *Exception* (página 190), directa o indirectamente. Por ejemplo:

```
>>> class MiError(Exception):
...     def __init__(self, valor):
...         self.valor = valor
...     def __str__(self):
...         return repr(self.valor)
...
>>> try:
...     raise MiError(2*2)
... except MiError as e:
...     print u'Ha ocurrido mi excepción, valor:', e.valor
...
Ocurrió mi excepción, valor: 4
>>> raise MiError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MiError: 'oops!'
```

En este ejemplo, el método `__init__()` de *Exception* (página 190) fue sobrescrito. El nuevo comportamiento simplemente crea el atributo *valor*.

Esto reemplaza el comportamiento por defecto de crear el atributo *args*.

Las clases de Excepciones pueden ser definidas de la misma forma que cualquier otra clase, pero usualmente se mantienen simples, a menudo solo ofreciendo un número de atributos con información sobre el error que leerán los manejadores de la excepción. Al crear un módulo que puede lanzar varios errores distintos, una práctica común es crear una clase base para excepciones definidas en ese módulo y extenderla para crear clases excepciones específicas para distintas condiciones de error:

```
class Error(Exception):
    """Clase base para excepciones en el módulo."""
    pass

class EntradaError(Error):
    """Exception lanzada por errores en las entradas.

    Atributos:
        expresion -- expresión de entrada en la que ocurre el error
        mensaje -- explicación del error
    """

    def __init__(self, expresion, mensaje):
        self.expresion = expresion
        self.mensaje = mensaje

class TransicionError(Error):
    """Lanzada cuando una operación intenta una
       transición de estado no permitida.
```

(continué en la próxima página)

(proviene de la página anterior)

```
    Atributos:
        previo -- estado al principio de la transición
        siguiente -- nuevo estado intentado
        mensaje -- explicación de porque la transición no esta permitida
    """
    def __init__(self, previo, siguiente, mensaje):
        self.previo = previo
        self.siguiente = siguiente
        self.mensaje = mensaje
```

La mayoría de las excepciones son definidas con nombres que terminan en «Error», similares a los nombres de las excepciones estándar.

Muchos módulos estándar definen sus propias excepciones para reportar errores que pueden ocurrir en funciones propias. Se puede encontrar más información sobre clases en el capítulo *Clases* (página 194).

9.1.7 Definiendo acciones de limpieza

La sentencia `try` tiene otra sentencia opcional que intenta definir acciones de limpieza que deben ser ejecutadas bajo ciertas circunstancias. Por ejemplo:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Adiós, Mundo!'
...
Chau, Mundo!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
```

Una *sentencia finally* siempre es ejecutada antes de salir de la sentencia `try`, ya sea que una excepción haya ocurrido o no. Cuando ocurre una excepción en la sentencia `try` y no fue manejada por una sentencia `except` (o ocurrió en una sentencia `except` o `else`), es relanzada luego de que se ejecuta la sentencia `finally`. La sentencia `finally` es también ejecutada «a la salida» cuando cualquier otra sentencia de la sentencia `try` es dejada vía `break`, `continue` or `return`. Un ejemplo más complicado (sentencias `except` y `finally` en la misma sentencia `try`):

```
>>> def dividir(x, y):
...     try:
...         resultado = x / y
...     except ZeroDivisionError:
...         print "¡división por cero!"
...     else:
...         print "el resultado es", resultado
...     finally:
...         print "ejecutando la clausula finally"
...
>>> dividir(2, 1)
el resultado es 2
ejecutando la clausula finally
>>> dividir(2, 0)
¡división por cero!
ejecutando la clausula finally
>>> divide("2", "1")
ejecutando la clausula finally
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Como puedes ver, la sentencia `finally` es ejecutada siempre. La excepción *TypeError* (página 193) lanzada al dividir dos cadenas de caracteres no es manejado por la sentencia `except` y por lo tanto es relanzada luego de que se ejecuta la sentencia `finally`.

En aplicaciones reales, la sentencia `finally` es útil para liberar recursos externos (como archivos o conexiones de red), sin importar si el uso del recurso fue exitoso.

9.1.8 Acciones predefinidas de limpieza

Algunos objetos definen acciones de limpieza estándar que llevar a cabo cuando el objeto no es más necesitado, independientemente de que las operaciones sobre el objeto hayan sido exitosas o no. Mirá el siguiente ejemplo, que intenta *abrir un archivo* (página 154) e imprimir su contenido en la pantalla.

```
for linea in open("numeros.txt"):
    print linea
```

El problema con este código es que deja el archivo abierto por un periodo de tiempo indeterminado luego de que termine de ejecutarse. Esto no es un problema en scripts simples, pero puede ser un problema en aplicaciones más grandes.

Sentencia with

La sentencia `with` permite que objetos como archivos sean usados de una forma que asegure que siempre se los libera rápido y en forma correcta.

```
with open("numeros.txt") as f:
    for linea in f:
        print linea
```

Luego de que la sentencia sea ejecutada, el archivo *f* siempre es cerrado, incluso si se encuentra un problema al procesar las líneas. Otros objetos que provean acciones de limpieza predefinidas lo indicarán en su documentación.

9.1.9 Traceback

El `Traceback` o *trazado inverso*, es un listado de las funciones en curso de ejecución, presentadas cuando sucede un error en tiempo de ejecución. Es común que al trazado inverso también se le conozca como *trazado de pila*, porque lista las funciones en el orden en el cual son almacenadas en la pila de llamadas[96].

El módulo integrado traceback[97] incorpora el comportamiento de `Traceback` o *trazado inverso* ya que extrae, formatea e imprime información acerca de *trazado del stack* de los errores y excepciones en Python.

```
>>> import traceback
>>> traceback.__doc__
'Extract, format and print information about Python stack traces.'
>>> help(traceback)
```

Importante: Usted puede descargar el código usado en esta sección haciendo clic en los siguientes enlaces: `excepciones_integradas.py`, `excepciones_propias.py` y `errores_propios.py`.

Truco: Para ejecutar el código `excepciones_integradas.py` y `errores_propios.py`, abra una consola de comando, acceda al directorio donde se encuentra ambos programas:

[96] https://es.wikipedia.org/wiki/Pila_(estructura_de_datos)#Pila_de_llamadas
[97] https://docs.python.org/library/traceback

```
leccion9/
├── excepciones_integradas.py
├── excepciones_propias.py
└── errores_propios.py
```

Si tiene la estructura de archivo previa, entonces ejecute el siguiente comando:

```
python excepciones_integradas.py
python errores_propios.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

9.2 Excepciones integradas

Las excepciones pueden ser objetos de una clase u objetos cadena. Aunque la mayoría de las excepciones eran objetos cadena en las anteriores versiones de Python, en Python 1.5 y versiones posteriores, todas las excepciones estándar han sido convertidas en objetos de clase y se anima a los usuarios a que hagan lo propio. Las excepciones están definidas en el módulo `exceptions`. Nunca es necesario importar este módulo explícitamente, pues las excepciones vienen proporcionadas por el espacio nominal interno.

Dos objetos de cadena distintos con el mismo valor se consideran diferentes excepciones. Esto es así para forzar a los programadores a usar nombres de excepción en lugar de su valor textual al especificar gestores de excepciones. El valor de cadena de todas las excepciones internas es su nombre, pero no es un requisito para las *excepciones definidas por el usuario* (página 187) u otras excepciones definidas por módulos de biblioteca.

En el caso de las clases de excepción, en una sentencia `try` con una sentencia `except` que mencione una clase particular, esta sentencia también gestionará cualquier excepción derivada de dicha clase (pero no las clases de excepción de las que deriva ella). Dos clases de excepción no emparentadas mediante sub-clasificación nunca son equivalentes, aunque tengan el mismo nombre.

Las excepciones internas enumeradas a continuación pueden ser generadas por el intérprete o por funciones internas. Excepto en los casos mencionados, tienen un `valor asociado` indicando en detalle la causa del error. Este valor puede ser una cadena o tupla de varios elementos informativos (es decir, un código de error y una cadena que explica el código). El valor asociado es el segundo argumento a la sentencia `raise`. En las cadenas de excepción, el propio valor asociado se almacenará en la variable nombrada como el segundo argumento de la sentencia `except` (si la hay). En las clases de excepción, dicha variable recoge la instancia de la excepción. Si la clase de excepción deriva de la clase raíz estándar *Exception* (página 190), el valor asociado está disponible en el atributo `args` de la instancia de excepción y es probable que aparezca en otros atributos.

El código de usuario puede lanzar excepciones internas. Se puede usar para comprobar un gestor de excepciones o para informar de una condición de error del mismo modo que el intérprete lanza la misma excepción. Hay que ser precavido, pues nada incluye que el código de usuario lance una excepción inadecuada.

Las siguientes excepciones sólo se usan como clase base de otras excepciones.

`BaseException` La clase base de todas las comunes excepciones. Deriva de la clase raíz `__builtin__.object`, es el tipo más básico.

`Exception` La clase base de todas las excepciones no existentes. Deriva de la clase raíz *BaseException* (página 190).

La clase base de las excepciones. Todas las excepciones internas derivan de esta clase. Todas las excepciones definidas por usuario deberían derivarse de esta clase, aunque no es obligatorio (todavía). La función `str()`, aplicada a una instancia de una clase (o la mayoría de sus clases derivadas) devuelve un valor cadena a partir de sus argumentos o una cadena vacía si no se proporcionaron argumentos al constructor. Si se

usa como secuencia, accede a los argumentos proporcionados al constructor (útil para compatibilidad con código antiguo). Los argumentos también están disponibles en el atributo `args` de la instancia, como tupla.

StandardError La clase base para todas las excepciones internas excepto *SystemExit* (página 192). Deriva de la clase raíz *Exception* (página 190).

ArithmeticError La clase base de las excepciones lanzadas por diversos errores aritméticos: *OverflowError* (página 192), *ZeroDivisionError* (página 193) y *FloatingPointError* (página 191). Deriva de la clase raíz *StandardError* (página 191).

LookupError La clase base de las excepciones lanzadas cuando una clave o índice utilizado en una correspondencia (diccionario) o secuencia son incorrectos: *IndexError* (página 191), *KeyError* (página 192). Deriva de la clase raíz *StandardError* (página 191).

EnvironmentError La clase base de las excepciones que pueden ocurrir fuera del sistema Python: *IOError* (página 191), *OSError* (página 192). Cuando se crean excepciones de este tipo con una tupla de dos valores, el primer elemento queda disponible en el atributo `errno` de la instancia (se supone que es un número de error) y el segundo en el atributo `strerror` (suele ser el mensaje de error asociado). La propia tupla está disponible en el atributo `args`. Cuando se instancia una excepción `EnvironmentError` con una tupla de tres elementos, los primeros dos quedan disponibles como en el caso de dos elementos y el tercero queda en el atributo `filename`. Sin embargo, por compatibilidad con sistemas anteriores, el atributo `args` contiene sólo una tupla de dos elementos de los dos primeros argumentos del constructor. El atributo `filename` es *None* (página 210) cuando se cree la excepción con una cantidad de argumentos diferente de 3. Los atributos `errno` y `strerror` son también `None` cuando la instancia no se cree con 2 ó 3 argumentos. En este último caso, `args` contiene los argumentos del constructor tal cual, en forma de tupla. Deriva de la clase raíz *StandardError* (página 191).

Las siguientes excepciones son las realmente lanzadas.

AssertionError Se lanza cuando una sentencia `assert` es `False`. Deriva de la clase raíz *StandardError* (página 191).

AttributeError Se lanza cuando una referencia o asignación a atributo fracasa (cuando un objeto no tenga referencias o asignaciones a atributos en absoluto, se lanza, la excepción *TypeError* (página 193).) Deriva de la clase raíz *StandardError* (página 191).

BufferError Se lanza cuando un error Buffer sucede. Deriva de la excepción *StandardError* (página 191).

EOFError Se lanza cuando las funciones internas (*input()* (página 123) o *raw_input()* (página 123)) alcanzan un *end of file* `EOF` (final de archivo) sin leer datos. N.B.: Los métodos *read()* (página 216) y *readline()* (página 217) de los objetos archivo devuelven una cadena vacía al alcanzar `EOF`. Deriva de la clase raíz *StandardError* (página 191).

FloatingPointError Se lanza cuando falla una operación de coma flotante. Esta excepción siempre está definida, pero sólo se puede lanzar cuando Python esta configurado con la opción `--with-fpectl` o se ha definido el símbolo `WANT_SIGFPE_HANDLER` en el archivo `config.h`. Deriva de la clase raíz *ArithmeticError* (página 191).

GeneratorExit Se lanza cuando la solicitud de salida de un generador Python sucede. Deriva de la excepción *BaseException* (página 190).

IOError Se lanza cuando una operación de E/S (tal como una sentencia *print* (página 153), la función integrada *open()* (página 119) o un método de un objeto archivo) fracasa por motivos relativos a E/S, por ejemplo, por no encontrarse un archivo o llenarse el disco. Esta clase se deriva de *EnvironmentError* (página 191). En la explicación anterior se proporciona información adicional sobre los atributos de instancias de excepción. Deriva de la clase raíz *EnvironmentError* (página 191).

ImportError Se lanza cuando una sentencia `import` no encuentra la definición del módulo o cuando `from ... import` no encuentra un nombre a importar. Deriva de la clase raíz *StandardError* (página 191).

IndexError Se lanza cuando un sub-índice de una secuencia se sale del rango. Los índices de corte se truncan silenciosamente al rango disponible. Si un índice no es un entero simple, se lanza *TypeError* (página 193). Deriva de la clase raíz *LookupError* (página 191).

IndentationError Se lanza cuando una indentación incorrecta sucede. Deriva de la excepción *SyntaxError* (página 192).

KeyError Se lanza cuando no se encuentra una clave de una correspondencia (diccionario) en el conjunto de claves existentes. Deriva de la clase raíz *LookupError* (página 191).

KeyboardInterrupt Se lanza cuando el usuario pulsa la tecla de interrupción (normalmente con la combinación de teclas `Control-C` o `DEL2.7`). A lo largo de la ejecución se comprueba si se ha interrumpido regularmente. Las interrupciones ocurridas cuando una función *input()* (página 123) o *raw_input()* (página 123)) espera datos también lanzan esta excepción. Deriva de la clase raíz *BaseException* (página 190).

MemoryError Se lanza cuando una operación agota la memoria pero aún se puede salvar la situación (borrando objetos). El valor asociado es una cadena que indica qué tipo de operación (interna) agotó la memoria. Obsérvese que por la arquitectura de gestión de memoria subyacente (la función de C `malloc()`), puede que el intérprete no siempre sea capaz de recuperarse completamente de esta situación. De cualquier modo, se lanza una excepción para que se pueda imprimir una traza, por si la causa fue un programa desbocado. Deriva de la clase raíz *StandardError* (página 191).

NameError Se lanza cuando no se encuentra un nombre local o global. Sólo se aplica a nombre no calificados. El valor asociado es el nombre no encontrado. Deriva de la clase raíz *StandardError* (página 191).

NotImplementedError Esta excepción se deriva de *RuntimeError* (página 192). En clases base definidas por el usuario, los métodos abstractos deberían lanzar esta excepción cuando se desea que las clases derivadas redefinan este método. Deriva de la clase raíz *RuntimeError* (página 192).

OSError Esta clase se deriva de *EnvironmentError* (página 191) y se usa principalmente como excepción os.error de os. En *EnvironmentError* (página 191) hay una descripción de los posibles valores asociados.

OverflowError Se lanza cuando el resultado de una operación aritmética es demasiado grande para representarse (desbordamiento). Esto no es posible en los enteros largos (que antes que rendirse lanzarían *MemoryError* (página 192)). Por la falta de normalización de la gestión de excepciones de coma flotante en C, la mayoría de las operaciones de coma flotante, tampoco se comprueban. En el caso de enteros normales, se comprueban todas las operaciones que pueden desbordar excepto el desplazamiento a la izquierda, en el que las aplicaciones típicas prefieren perder bits que lanzar una excepción. Deriva de la clase raíz *ArithmeticError* (página 191).

RuntimeError Se lanza cuando se detecta un error que no cuadra en ninguna de las otras categorías. El valor asociado es una cadena que indica qué fue mal concretamente. Esta excepción es mayormente una reliquia de versiones anteriores del intérprete; ya casi no se usa. Deriva de la clase raíz *StandardError* (página 191).

StopIteration Se lanza cuando se indica el final desde `iterator.next()`. Deriva de la excepción *Exception* (página 190).

SyntaxError Se lanza cuando el analizador encuentra un error en la sintaxis. Esto puede ocurrir en una sentencia `import`, en una sentencia `exec`, en una llamada a la función interna `eval()` o *input()* (página 123), o al leer el guion inicial o la entrada estándar (por ejemplo, la entrada interactiva). Si se usan excepciones de clase, las instancias de esta clase tienen disponibles los atributos `filename` (nombre del archivo), `lineno` (nº de línea), `offset` (nº de columna) y `text` (texto), que ofrecen un acceso más fácil a los detalles. En las excepciones de cadena, el valor asociado suele ser una tupla de la forma (mensaje, (nombreFichero, numLinea, columna, texto)). En las excepciones de clase, `str()` sólo devuelve el mensaje. Deriva de la clase raíz *StandardError* (página 191).

SystemError Se lanza cuando el intérprete encuentra un error interno, pero la situación no parece tan grave como para perder la esperanza. El valor asociado es una cadena que indica qué ha ido mal (en términos de bajo nivel). Se debería dar parte de este error al autor o mantenedor del intérprete Python en cuestión. Se debe incluir en el informe la cadena de versión del intérprete Python (`sys.version`, que también se muestra al inicio de una sesión interactiva), la causa exacta del error y, si es posible, el código fuente del programa que provocó el error. Deriva de la clase raíz *StandardError* (página 191).

SystemExit Lanzada por la función `sys.exit()`. Si no se captura, el intérprete de Python finaliza la ejecución sin presentar una pila de llamadas. Si el valor asociado es un entero normal, especifica el estado de salida al sistema (se pasa a la función de C `exit()`), Si es None, el estado de salida es cero (que indica una salida normal sin errores). En el caso de ser de otro tipo, se presenta el valor del objeto y el estado de salida será `1`. Las instancias tienen un atributo `code` cuyo valor se establece al estado de salida o mensaje de error propuesto (inicialmente `None`). Además, esta excepción deriva directamente de *Exception* (página 190) y no de la excepción *StandardError* (página 191), ya que técnicamente no es un error. Una llamada a `sys.exit()` se traduce a un error para que los gestores de limpieza final (las sentencias `finally` de las

sentencias `try`) se puedan ejecutar y para que un depurador pueda ejecutar un guion sin riesgo de perder el control. Se puede usar la función os._exit() si es total y absolutamente necesario salir inmediatamente (por ejemplo, tras un `fork()` en el proceso hijo). Deriva de la clase raíz *BaseException* (página 190).

ReferenceError Se lanza cuando se usó un proxy de referencia débil después de que el referente desapareció. Deriva de la excepción *StandardError* (página 191).

TabError Se lanza cuando sucede una mezcla inadecuada de espacios y tabulaciones. Deriva de la excepción *IndentationError* (página 191).

TypeError Se lanza cuando una operación o función interna se aplica a un objeto de tipo inadecuado. El valor asociado es una cadena con detalles de la incoherencia de tipos. Deriva de la clase raíz *StandardError* (página 191).

UnboundLocalError Se lanza cuando se hace referencia a una variable local en una función o método, pero no se ha asignado un valor a dicha variable. Deriva de la excepción *NameError* (página 192).

UnicodeError Se lanza cuando se da un error relativo a codificación/descodificación `Unicode`. Deriva de la excepción *ValueError* (página 193).

UnicodeDecodeError Se lanza cuando un error al decodificar `Unicode` sucede. Deriva de la excepción *UnicodeError* (página 193).

UnicodeEncodeError Se lanza cuando un error al codificar `Unicode` sucede. Deriva de la excepción *UnicodeError* (página 193).

UnicodeTranslateError Se lanza cuando un error al traducir `Unicode` sucede. Deriva de la excepción *UnicodeError* (página 193).

ValueError Se lanza cuando una operación o función interna recibe un argumento del tipo correcto, pero con un valor inapropiado y no es posible describir la situación con una excepción más precisa, como *IndexError* (página 191).

ZeroDivisionError Se lanza cuando el segundo argumento de una operación de división o módulo es cero. El valor asociado es una cadena que indica el tipo de operandos y la operación. Deriva de la clase raíz *ArithmeticError* (página 191).

Warning La clase base para las categorías de advertencias. Deriva de la excepción *Exception* (página 190).

BytesWarning La clase base para las advertencias acerca de problemas relacionados con bytes y buffer, mas relacionado a la conversión desde `str` o comparando a `str`. Deriva de la excepción *Warning* (página 193).

DeprecationWarning La clase base para las advertencias acerca de características obsoletas. Deriva de la excepción *Warning* (página 193).

FutureWarning La clase base para las advertencias acerca de constructores que pueden ser cambiado sistemáticamente en el futuro. Deriva de la excepción *Warning* (página 193).

ImportWarning La clase base para las advertencias acerca de probables errores en importar módulos. Deriva de la excepción *Warning* (página 193).

PendingDeprecationWarning La clase base para las advertencias acerca de características las cuales serán obsoletas en el futuro. Deriva de la excepción *Warning* (página 193).

RuntimeWarning La clase base para las advertencias acerca de comportamiento del tiempo de ejecución dudosa. Deriva de la excepción *Warning* (página 193).

SyntaxWarning La clase base para las advertencias acerca de sintaxis dudosa. Deriva de la excepción *Warning* (página 193).

UnicodeWarning La clase base para las advertencias acerca de problemas relacionado con `Unicode`, más relacionado a problemas de conversión. Deriva de la excepción *Warning* (página 193).

UserWarning La clase base para las advertencias generadas por código de usuario. Deriva de la excepción *Warning* (página 193).

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

9.3 Programación orientada a objetos

La programación orientada a objetos (POO, u OOP según sus siglas en inglés) es un paradigma de programación que viene a innovar la forma de obtener resultados. Los objetos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad especial.

Muchos de los objetos prediseñados de los lenguajes de programación actuales permiten la agrupación en bibliotecas o librerías, sin embargo, muchos de estos lenguajes permiten al usuario la creación de sus propias bibliotecas.

Figura 9.1: Programación Orientada a Objetos - POO

Está basada en varias técnicas, como las siguientes:

- *herencia* (página 202).
- cohesión.
- *abstracción* (página 207).
- *polimorfismo* (página 208).
- acoplamiento.
- *encapsulación* (página 207).

La POO tiene sus raíces en la década del 60 con el lenguaje de programación `Simula` que en 1967, el cual fue el primer lenguaje que posee las características principales de un lenguaje orientado a objetos.

Smalltalk (de 1972 a 1980) es posiblemente el ejemplo canónico, y con el que gran parte de la teoría de la POO se ha desarrollado. Más su uso se popularizó a principios de la década de 1990.

En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos.

Los objetivos de la POO son:

- Organizar el código fuente, y
- re-usar código fuente en similares contextos.

Nota: Más información consulte el articulo de Wikipedia Programación orientada a objetos[98].

---

[98] https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

9.3.1 POO en Python

El mecanismo de clases de Python agrega clases al lenguaje con un mínimo de nuevas sintaxis y semánticas.

En Python las clases es una mezcla de los mecanismos de clase encontrados en C++ y Modula-3.

Como es cierto para los módulos, las clases en Python no ponen una barrera absoluta entre la definición y el usuario, sino que más bien se apoya en la cortesía del usuario de no «forzar la definición».

Sin embargo, se mantiene el poder completo de las características más importantes de las clases: el mecanismo de la herencia de clases permite múltiples clases base, una clase derivada puede sobrescribir cualquier método de su(s) clase(s) base, y un método puede llamar al método de la clase base con el mismo nombre.

> *«Los objetos pueden tener una cantidad arbitraria de datos.»*

En terminología de C++, todos los miembros de las clases (incluyendo los miembros de datos), son *públicos*, y todas las funciones miembro son *virtuales*.

Como en Modula-3, no hay atajos para hacer referencia a los miembros del objeto desde sus métodos: la función método se declara con un primer argumento explícito que representa al objeto, el cual se provee implícitamente por la llamada.

Como en Smalltalk, las clases mismas son objetos. Esto provee una semántica para importar y renombrar.

A diferencia de C++ y Modula-3, los tipos de datos integrados pueden usarse como clases base para que el usuario los extienda.

También, como en C++ pero a diferencia de Modula-3, la mayoría de los operadores integrados con sintaxis especial (operadores aritméticos, de subíndice, etc.) pueden ser redefinidos por instancias de la clase.

(Sin haber una terminología universalmente aceptada sobre clases, haré uso ocasional de términos de Smalltalk y C++. Usaría términos de Modula-3, ya que su semántica orientada a objetos es más cercana a Python que C++, pero no espero que muchos lectores hayan escuchado hablar de él).

Algunas particularidades de POO en Python son las siguientes:

- Todo es un objeto, incluyendo los tipos y clases.
- Permite herencia múltiple.
- No existen métodos ni atributos privados.
- Los atributos pueden ser modificados directamente.
- Permite «monkey patching».
- Permite «duck typing».
- Permite la sobrecarga de operadores.
- Permite la creación de nuevos tipos de datos.

A continuación se procede a definir algunos conceptos necesarios para entender la POO:

9.3.2 Objetos

Los *objetos* son abstracción de Python para data. Toda la data en un programa Python es representado por objectos o por relaciones entre objectos. (En cierto sentido, y en el código modelo de Von Neumann de una «computadora almacenada del programa» también es un código representado por los objetos.)

Cada objeto tiene una identidad, un tipo y un valor. Una identidad de objecto nunca cambia una vez es creada; usted puede pensar eso como la dirección de objeto en memoria. El operador *in* (página 84) compara la identidad de dos objetos; la función *id()* (página 118) devuelve un número entero representando la identidad (actualmente implementado como su dirección).

El *tipo* de un objeto también es inmutable. El tipo de un objeto determina las operaciones que admite el objeto (por ejemplo, «¿tiene una longitud?») Y también define los valores posibles para los objetos de ese tipo. La función «*type()* (página 122)» devuelve el tipo de un objeto (que es un objeto en sí mismo). El *valor *de algunos*

*objetos puede cambiar. Se dice que los objetos cuyo valor puede cambiar son *mutables*; los objetos cuyo valor no se puede cambiar una vez que se crean se llaman *immutable*. (El valor de un objeto contenedor inmutable que contiene una referencia a un objeto mutable puede cambiar cuando se cambia el valor de este último; sin embargo, el contenedor todavía se considera inmutable, porque la colección de objetos que contiene no se puede cambiar. Por lo tanto, la inmutabilidad no es estrictamente lo mismo que tener un valor incambiable, es más sutil.) La mutabilidad de un objeto está determinada por su tipo; por ejemplo, los números, las cadenas y las tuplas son inmutables, mientras que los diccionarios y las listas son mutables.

Los objetos son la clave para entender la *POO* (página 194). Si mira a nuestro alrededor encontrará un sin fin de objetos de la vida real: perro, escritorio, televisor, bicicleta, etc…

En Python puede definir una clase con la palabra reservada *class* (página 200), de la siguiente forma:

```
class Persona:
    pass
```

En el ejemplo anterior, el nombre de la clase es `Persona` y dentro del bloque de código usa la sentencia *pass* (página 104). Aunque no es requerido por el intérprete, los nombres de las clases se escriben por convención capitalizadas. Las clases pueden (y siempre deberían) tener comentarios.

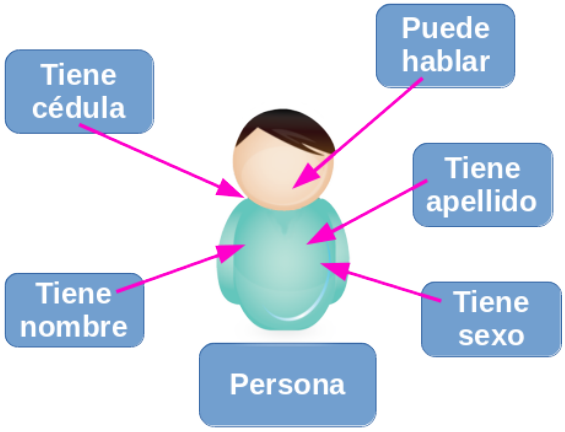

Figura 9.2: Diagrama de Objeto Persona

Estado de un objeto

El conjunto de datos y objetos relacionados con un objeto en un momento dado, se le conoce como «estado». Un objeto puede tener múltiples estados a lo largo de su existencia conforme se relaciona con su entorno y otros objetos.

Por hacer: TODO explicar el concepto Estado de un objeto.

9.3.3 Atributos

Los atributos o propiedades de los objetos son las características que puede tener un objeto, como el color. Si el objeto es `Persona`, los atributos podrían ser: `cedula`, `nombre`, `apellido`, `sexo`, etc…

Los atributos describen el estado de un objeto. Pueden ser de cualquier tipo de dato.

```
class Persona:
    """Clase que representa una Persona"""
    cedula = "V-13458796"
    nombre = "Leonardo"
```

(continué en la próxima página)

(proviene de la página anterior)

```
    apellido = "Caballero"
    sexo = "M"
```

Usted puede probar el código anterior, si lo transcribe en el *consola interactiva* (página 15) Python como lo siguiente:

```
>>> class Persona:
...     """Clase que representa una Persona"""
...     cedula = "V-13458796"
...     nombre = "Leonardo"
...     apellido = "Caballero"
...     sexo = "M"
...
>>> macagua = Persona
>>> type(macagua)
<type 'classobj'>
>>> dir(macagua)
['__doc__', '__module__', 'apellido', 'cedula', 'nombre', 'sexo']
>>> macagua.cedula
'V-13458796'
>>> macagua.nombre
'Leonardo'
>>> macagua.apellido
'Caballero'
>>> macagua.sexo
'M'
>>> print "El objeto de la clase " + macagua.__name__ +"," \
... + macagua.__doc__ + "."
El objeto de la clase Persona, Clase que representa una Persona.
>>> print "Hola, mucho gusto, mi nombre es '"+ \
... macagua.nombre +" "+ \
... macagua.apellido +"', \nmi cédula de identidad es '"+  \
... macagua.cedula +"', y mi sexo es '"+  \
... macagua.sexo +"'."
Hola, mucho gusto, mi nombre es 'Leonardo Caballero',
mi cédula de identidad es 'V-13458796', y mi sexo es 'M'.
```

Si el nombre de un atributo esta encerrado entre dobles guiones bajos son atributos especiales.

- __name__, describe el nombre del objeto o del método.

```
>>> macagua.__name__
'Persona'
```

- __doc__, contiene la documentación de un módulo, una clase, o método especifico, escrita en el formato *docstrings* (página 50).

```
>>> macagua.__doc__
'Clase que representa una Persona'
```

Si el nombre de un atributo esta con dobles guiones bajos al principio son atributos «escondidos». A continuación un pseudo código que ilustra un ejemplo:

```
>>> ms_windows.__privado
'True'

>>> ms_windows.codigo_fuente.__no_tocar
'True'
```

En la sección *encapsulación* (página 207) se describe esto a más profundidad.

9.3.4 Métodos

Los métodos describen el comportamiento de los objetos de una clase. Estos representan las operaciones que se pueden realizar con los objetos de la clase,

La ejecución de un método puede conducir a cambiar el estado del objeto.

Se definen de la misma forma que las funciones normales pero deben declararse dentro de la clase y su primer argumento siempre referencia a la instancia que la llama, de esta forma se afirma que los métodos son *funciones* (página 100), adjuntadas a *objectos* (página 195).

Nota: Usted puede encontrar ejemplos en las funciones de *cadena de caracteres* (página 46), *listas* (página 58), *diccionarios* (página 65), etc.

Si el objeto es `Persona`, los métodos pueden ser: `hablar`, `caminar`, `comer`, `dormir`, etc.

```
class Persona:
    """Clase que representa una Persona"""
    cedula = "V-13458796"
    nombre = "Leonardo"
    apellido = "Caballero"
    sexo = "M"

    def hablar(self, mensaje):
        """Mostrar mensaje de saludo de Persona"""
        return mensaje
```

La única diferencia sintáctica entre la definición de un método y la definición de una función es que el primer parámetro del método por convención debe ser el nombre `self`.

Usted puede probar el código anterior, si lo transcribe en el *consola interactiva* (página 15) Python como lo siguiente:

```
>>> class Persona:
...     """Clase que representa una Persona"""
...     cedula = "V-13458796"
...     nombre = "Leonardo"
...     apellido = "Caballero"
...     sexo = "M"
...
...     def hablar(self, mensaje):
...         """Mostrar mensaje de saludo de Persona"""
...         return mensaje
...
>>>
>>> macagua = Persona
>>> Persona().hablar("Hola, soy la clase {0}.".format(
...     macagua.__name__))
'Hola, soy la clase Persona.'
>>> type(Persona().hablar)
<type 'instancemethod'>
>>> Persona().hablar.__doc__
'Mostrar mensaje de saludo de Persona'
```

Si crea una instancia de objeto para la clase `Persona` e intenta llamar al método `hablar()` esto lanzara una excepción *TypeError* (página 193), como sucede a continuación:

```
>>> macagua = Persona
>>> macagua.hablar("Hola Plone")
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  File "<stdin>", line 1, in <module>
TypeError: unbound method hablar() must be called with Persona instance as first␣
↪argument (got str instance instead)
```

Esto sucede por...

Por hacer: TODO explicar por que se lanza la excepción TypeError.

Ámbito de los métodos

Los métodos cuentan con un espacio de nombres propio. En caso de no encontrar un nombre en su ámbito local, buscará en el ámbito superior hasta encontrar alguna coincidencia.

Los métodos pueden acceder y crear atributos dentro del objeto al que pertenecen, anteponiendo la palabra `self` y el operador de atributo «.» antes del nombre del atributo en cuestión.

Métodos especiales

Las clases en Python cuentan con múltiples métodos especiales, los cuales se encuentran entre dobles guiones bajos `__<metodo>__()`.

Los métodos especiales más utilizados son *__init__()* (página 201), *__str__()* (página 199) y *__del__()* (página 199).

__str__()

El método `__str__()` es un método especial, el cual se ejecuta al momento en el cual un objeto se manda a mostrar, es decir es una cadena representativa de la clase, la cual puede incluir formatos personalizados de presentación del mismo.

```
    def __str__(self):
        """Devuelve una cadena representativa de Persona"""
        return "%s: %s, %s %s, %s." % (
            self.__doc__[25:34], str(self.cedula), self.nombre,
            self.apellido, self.getGenero(self.sexo))
```

__del__()

El método `__del__()` es un método especial, el cual se ejecuta al momento en el cual un objeto es descartado por el intérprete. El comportamiento de `__del__()` es muy similar a los «destructores» en otros lenguajes.

Métodos de clase

En ocasiones es necesario contar con métodos que interactúen con elementos de la clase de la cual el objeto es instanciado. Python permite definir métodos de clase para esto.

Los métodos de clase son aquellos que están ligados directamente con los atributos definidos en la clase que los contiene. Para definir un método de clase se utiliza el decorador `@classmethod` y por convención se utiliza `cls` como argumento inicial en lugar de `self`.

Del mismo modo, los métodos de clase utilizan el prefijo `cls` para referirse a los atributos de la clase.

```
class <Clase>(object):
    ...
    ...
    @classmethod
    def <metodo>(cls, <argumentos>):
        ...
        ...
```

Métodos estáticos

Los métodos estáticos hacen referencia a las instancias y métodos de una clase. Para definir un método estático se utiliza el decorador `@staticmethod` y no utiliza ningún argumento inicial.

Al no utilizar `self`, los métodos estáticos no pueden interactuar con los atributos y métodos de la instancia.

Para referirse a los elementos de la clase, se debe utilizar el nombre de la clase como prefijo.

```
class <Clase>(object):
    ...
    ...
    @staticmethod
    def <metodo>(<argumentos>):
        ...
        ...
```

Interfaces

La forma en que los métodos de un objeto pueden ser accedidos por otros objetos se conoce como «interfaz». Una interfaz bien definida permite a objetos de distinta índole interactuar entre sí de forma modular. La interfaz define el modo en que los objetos intercambian información.

Por hacer: TODO explicar el concepto de Interfaces.

Implementaciones

Una implementación corresponde al mecanismo interno que se desencadena en un método cuando éste es llamado. Las implementaciones procesan las entradas proveniente de las interfaces y actúan en consecuencia ya sea:

- Modificando el estado del objeto.
- Transfiriendo la información resultante del proceso interno a través de la interfaces.

Por hacer: TODO explicar el concepto de Implementaciones.

9.3.5 Clases

Las clases definen las características del *objeto* (página 195).

Con todos los conceptos anteriores explicados, se puede decir que una clase es una plantilla genérica de un *objeto* (página 195). La clase proporciona variables iniciales de estado (donde se guardan los *atributos* (página 196)) e implementaciones de comportamiento (*métodos* (página 198)) necesarias para crear nuevos objetos, son los modelos sobre los cuáles serán construidos.

9.3.6 Instancias

Ya sabe que una clase es una estructura general del objeto. Por ejemplo, puede decir que la clase `Persona` necesita tener una `cedula`, un `nombre`, un `apellido` y una `sexo`, pero no va a decir cual es `cedula`, `nombre`, `apellido` y `sexo`, es aquí donde entran las instancias.

Una instancia es una copia específica de la clase con todo su contenido. Por ejemplo:

```
>>> persona1 = Persona("V-13458796", "Leonardo", "Caballero", "M")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: this constructor takes no arguments
```

La excepción *TypeError* (página 193) indica que el *método constructor no toma argumentos*, esto se debe a que la momento de definir la clase a cada atributo se le asigno un valor (tipo de dato).

Usted puede definir el metodo constructor de la clase usando el método *__init__()* (página 201).

Método __init__()

El método `__init__()` es un método especial, el cual se ejecuta al momento de instanciar un objeto. El comportamiento de `__init__()` es muy similar a los **«constructores»** en otros lenguajes. Los argumentos que se utilizan en la definición de `__init__()` corresponden a los parámetros que se deben ingresar al instanciar un objeto.

```
    def __init__(self, cedula, nombre, apellido, sexo):
        """Constructor de clase Persona"""
        self.cedula = cedula
        self.nombre = nombre
        self.apellido = apellido
        self.sexo = sexo
```

Función isinstance()

`isinstance()`, es una *función integrada* (página 113) la cual le permite corroborar si un objeto es instancia de una clase.

Nota: Más información consulte la documentación detallada de la función *isinstance()* (página 142).

Importante: Usted puede descargar el código usado en esta sección haciendo clic `aquí`.

Truco: Para ejecutar el código `poo.py`, abra una consola de comando, acceda al directorio donde se encuentra el mismo, y ejecute el siguiente comando:

```
python poo.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

9.4 Herencia

La herencia es una de las premisas y técnicas de la *POO* (página 194) la cual permite a los programadores crear una clase general primero y luego más tarde crear clases más especializadas que re-utilicen código de la clase general. La herencia también le permite escribir un código más limpio y legible.

9.4.1 Clase Base

Clase Base o también conocida como *Clase abstracta* le permite definir una clase que puede heredarse en otras clases los atributos y comportamientos definido en esta.

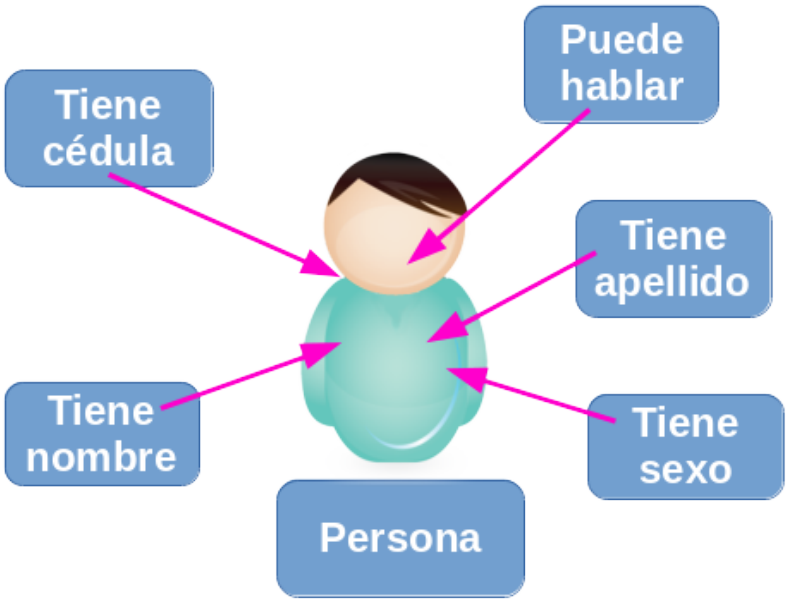

Figura 9.3: Clase base o abstracta

Use el diagrama anterior para ilustrar el concepto de la herencia, vea el caso de dos clases que tiene algo en común, ambas son personas, con atributos de datos personales y comportamiento típicos como hablar, comer, caminar, entonces para eso se crea una clase base llamada `Persona`. A continuación un ejemplo de la clase `Persona` con un método interno:

```
class Persona(object):
    """Clase que representa una Persona"""

    def __init__(self, cedula, nombre, apellido, sexo):
        """Constructor de clase Persona"""
        self.cedula = cedula
        self.nombre = nombre
        self.apellido = apellido
        self.sexo = sexo

    def __str__(self):
        """Devuelve una cadena representativa de Persona"""
        return "%s: %s, %s %s, %s." % (
            self.__doc__[25:34], str(self.cedula), self.nombre,
            self.apellido, self.getGenero(self.sexo))

    def hablar(self, mensaje):
        """Mostrar mensaje de saludo de Persona"""
        return mensaje
```

(continué en la próxima página)

(proviene de la página anterior)

```
    def getGenero(self, sexo):
        """Mostrar el genero de la Persona"""
        genero = ('Masculino','Femenino')
        if sexo == "M":
            return genero[0]
        elif sexo == "F":
            return genero[1]
        else:
            return "Desconocido"
```

En el ejemplo previo, es donde empieza a crear una clase (lo hace con la palabra `class`). La segunda palabra `Persona` es el nombre de la clase. La tercera palabra que se encuentra dentro de los paréntesis este hace referencia al objeto *object* (página 224), usando para indicar la clase de la cual precede.

La clase `Persona` tiene los métodos `__init__`, `__str__`, `hablar` y `getGenero`. Sus atributos son `cedula`, `nombre`, `apellido` y `sexo`.

La instancia de dos nuevos objetos `Persona` seria de la siguiente forma:

```
persona1 = Persona("V-13458796", "Leonardo", "Caballero", "M")
persona2 = Persona("V-23569874", "Ana", "Poleo", "F")
```

El método constructor `__init__` es un método especial el cual debe escribir como: `MiClase(parámetros iniciales si hay cualquiera)`.

Usted puede llamar esos métodos y atributos con la siguiente notación: `claseinstancia.metodo` o `claseinstancia.atributo`.

```
>>> print persona1.nombre, persona1.apellido
>>> print persona1.getGenero(persona1.sexo)
```

El método `__str__` es un método usando para imprimir la descripción de la instancia de objeto el cual debe mostrar como:

```
print "\n" + str(persona1) + "\n"
```

En el anterior código se usan para cierto formato para imprimir la instancia de objeto usando la sentencia `print`, concatenando el carácter `\n` para generar un salto de página y seguidamente convertir a formato cadena de caracteres usando la función `str()` a la instancia de objeto llamada `persona2`.

9.4.2 Herencia simple

La herencia simple se apoya en el uso de *clases base* (página 202) para compartir sus atributos y comportamientos con otros clases derivadas como los siguiente ejemplos el objeto `Supervisor` y el objeto `Obrero`.

El siguiente es un ejemplo de la clase `Supervisor` que derivada de la clase `Persona` con función interna:

```
class Supervisor(Persona):
    """Clase que representa a un Supervisor"""

    def __init__(self, cedula, nombre, apellido, sexo, rol):
        """Constructor de clase Supervisor"""

        # Invoca al constructor de clase Persona
        Persona.__init__(self, cedula, nombre, apellido, sexo)

        # Nuevos atributos
        self.rol = rol
        self.tareas = ['10','11','12','13']
```

(continué en la próxima página)

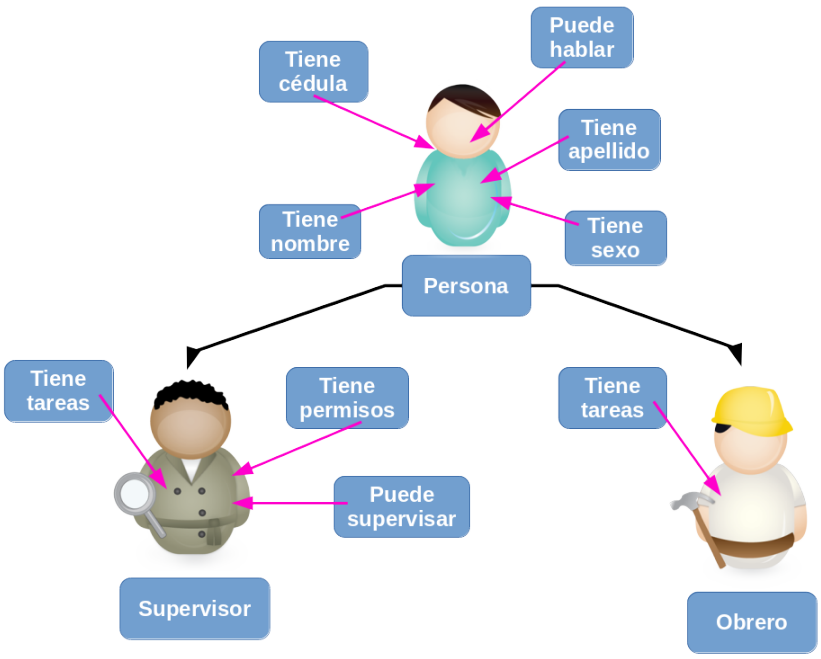

Figura 9.4: Diagrama de herencia de Objetos

(proviene de la página anterior)

```
    def __str__(self):
        """Devuelve una cadena representativa al Supervisor"""
        return "%s: %s %s, rol: '%s', sus tareas: %s." % (
            self.__doc__[26:37], self.nombre, self.apellido,
            self.rol, self.consulta_tareas())

    def consulta_tareas(self):
        """Mostrar las tareas del Supervisor"""
        return ', '.join(self.tareas)
```

Ahora, se creará una nueva clase `Supervisor` con los mismos métodos y atributos como la clase `Persona`, pero con dos nuevos atributos `rol` y `tareas`. No se copia la clase previa, pero si **se hereda** de ella.

La instancia del nuevo objeto `Supervisor` seria de la siguiente forma:

```
supervisor1 = Supervisor("V-16987456", "Jen", "Paz", "D", "Chivo")
```

Luego que generá la instancia del nuevo objeto `Supervisor` llamada `supervisor1` se puede imprimir sus detalles de la siguiente forma:

```
print "\n" + str(supervisor1) + "\n"
```

Como la instancia de objeto `supervisor1` hereda los atributo(s) y método(s) de la clase `Persona` usted puede reusarlo y llamarlo de la siguiente forma:

```
print "- Cedula de identidad: {0}.".format(supervisor1.cedula)
print "- Nombre completo: {0} {1}.".format(
        supervisor1.nombre, supervisor1.apellido)
print "- Genero: {0}.".format(
        supervisor1.getGenero(supervisor1.sexo))
print "- {0} {1} dijo: {2}".format(
        supervisor1.nombre, supervisor1.apellido,
        supervisor1.hablar("A trabajar Leonardo!!!".upper()))
```

Si desea usar los atributo(s) y método(s) heredados de la clase `Supervisor` se puede imprimir de la siguiente forma:

```
print "- Rol: {0}.".format(supervisor1.rol)
print "- N. Tareas: {0}.".format(supervisor1.consulta_tareas())
```

El uso de las clases y la programación orientada a objetos, le permite a usted que pueda organizar el código con diferentes clases correspondientes a diferentes objetos que encontrará (una clase `Persona`, una clase `Carro`, una clase `Departamento`, etc.), con sus propios métodos y atributos. Luego puede usar la herencia para considerar las variaciones en torno a una clase base y reutilizar el código. Ej.: a partir de una clase base de `Persona`, usted puede crear clases derivadas como `Supervisor`, `JefeCuadrilla`, `Obrero`, etc.

```
print """\nHola, Soy el {0} {1} {2}, mi cédula es '{3}',
mi genero '{4}', con el rol '{5}' y mis tareas
asignadas '{6}'.""".format(
    supervisor1.__doc__[26:37].lower(),
    supervisor1.nombre, supervisor1.apellido, supervisor1.cedula,
    supervisor1.getGenero(supervisor1.sexo), supervisor1.rol,
    supervisor1.consulta_tareas())
```

Función issubclass()

`issubclass()`, es una *función integrada* (página 113) la cual le permite corroborar si un objeto es instancia de una clase.

Nota: Más información consulte la documentación detallada de la función *issubclass()* (página 142).

Importante: Usted puede descargar el código usado en esta sección haciendo clic en los siguientes enlaces: `clases.py` y `herencia_simple.py`.

Truco: Para ejecutar el código `clases.py` y `herencia_simple.py`, abra una consola de comando, acceda al directorio donde se encuentra ambos programas:

```
leccion9/
├── clases.py
└── herencia_simple.py
```

Si tiene la estructura de archivo previa, entonces ejecute el siguiente comando:

```
python herencia_simple.py
```

9.4.3 Herencia múltiple

A diferencia de lenguajes como *Java* y *C#*, el lenguaje *Python* permite la herencia múltiple, es decir, se puede heredar de múltiples clases.

La herencia múltiple es la capacidad de una subclase de heredar de múltiples súper clases.

Esto conlleva un problema, y es que si varias súper clases tienen los mismos atributos o métodos, la subclase sólo podrá heredar de una de ellas.

En estos casos Python dará prioridad a las clases más a la izquierda en el momento de la declaración de la subclase:

```
class Destreza(object):
    """Clase la cual representa la Destreza de la Persona"""

    def __init__(self, area, herramienta, experiencia):
        """Constructor de clase Destreza"""
        self.area = area
        self.herramienta = herramienta
        self.experiencia = experiencia

    def __str__(self):
        """Devuelve una cadena representativa de la Destreza"""
        return """Destreza en el área %s con la herramienta %s,
        tiene %s años de experiencia.""" % (
            str(self.area), self.experiencia, self.herramienta)


class JefeCuadrilla(Supervisor, Destreza):
    """Clase la cual representa al Jefe de Cuadrilla"""

    def __init__(self, cedula, nombre, apellido, sexo,
        rol, area, herramienta, experiencia, cuadrilla):
        """Constructor de clase Jefe de Cuadrilla"""

        # Invoca al constructor de clase Supervisor
        Supervisor.__init__(self, cedula, nombre, apellido, sexo,
            rol)
        # Invoca al constructor de clase Destreza
        Destreza.__init__(self, area, herramienta, experiencia)

        # Nuevos atributos
        self.cuadrilla = cuadrilla

    def __str__(self):
        """Devuelve cadena representativa al Jefe de Cuadrilla"""
        jq = "{0}: {1} {2}, rol '{3}', tareas {4}, cuadrilla: {5}"
        return jq.format(
            self.__doc__[28:46], self.nombre, self.apellido,
            self.rol, self.consulta_tareas(), self.cuadrilla)
```

Method Resolution Order (MRO)

Ese es el orden en el cual el método debe heredar en la presencia de herencia múltiple. Usted puede ver el MRO usando el atributo `__mro__`.

```
>>> JefeCuadrilla.__mro__
(<class '__main__.JefeCuadrilla'>,
<class '__main__.Supervisor'>,
<class '__main__.Persona'>,
<class '__main__.Destreza'>,
<type 'object'>)
```

```
>>> Supervisor.__mro__
(<class '__main__.Supervisor'>,
<class '__main__.Persona'>,
<type 'object'>)
```

```
>>> Destreza.__mro__
(<class '__main__.Destreza'>,
<type 'object'>)
```

El *MRO* es calculado en Python de la siguiente forma:

Un método en la llamada derivada es siempre llamada antes de método de la clase base. En nuestro ejemplo, la clase `JefeCuadrilla` es llamada antes de las clases `Supervisor` o `Destreza`. Esas dos clases son llamada antes de la clase `Persona` y la clase `Persona` es llamada antes de la clase `object`.

Si hay herencia múltiple como `JefeCuadrilla(Supervisor, Destreza)`, el método invoca a `Supervisor` primero por que ese aparece primero de izquierda a derecha.

Importante: Usted puede descargar el código usado en esta sección haciendo clic en los siguientes enlaces: `clases.py` y `herencia_multiple.py`.

Truco: Para ejecutar el código `clases.py` y `herencia_multiple.py`, abra una consola de comando, acceda al directorio donde se encuentra ambos programas:

```
leccion9/
├── clases.py
└── herencia_multiple.py
```

Si tiene la estructura de archivo previa, entonces ejecute el siguiente comando:

```
python herencia_multiple.py
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

9.5 Abstracción

El concepto de encapsulamiento se apoya sobre el concepto de abstracción.

En *POO* (página 194) solo necesita saber como interaccionar con los objetos, no necesita conocer los detalles de cómo está implementada la clase a partir de la cual se instancia el objeto. Sólo necesita conocer su interfaz pública.

La encapsulación es una forma de abstracción, ademas es un mecanismo para llevar a la práctica la abstracción.

El nivel de abstracción puede ser bajo (en un objeto se manipulan datos y métodos individualmente), o alto (en un objeto solo se usan sus métodos de servicio).

9.5.1 Encapsulación

La encapsulación se considera una de las características definitorias de la *POO* (página 194).

Cuando una clase existe (se define), se crean objetos a partir de ella, y se usan dichos objetos llamando los métodos necesarios. Es decir, crea objetos para usar los servicios que nos proporciona la clase a través de sus métodos.

No necesita saber cómo trabaja el objeto, ni saber las variables que usa, ni el código que contiene.

El objeto es una *caja negra*. –> Modelo cliente - servidor. Es decir, el objeto es un servidor que proporciona servicios a los clientes que lo solicitan.

La encapsulación describe el hecho de que los objetos se usan como *cajas negras*. Así, un objeto encapsula datos y métodos, que están dentro del objeto.

Interfaz pública de una clase: Es el conjunto de métodos (métodos de servicio) que sirve para que los objetos de una clase proporcionen sus servicios. Estos servicios son los que pueden ser llamados por un cliente.

Métodos de soporte: Son métodos adicionales en un objeto que no definen un servicio utilizable por un cliente, pero que ayudan a otros métodos en sus tareas.

La encapsulación es un **mecanismo de control**. El estado (el conjunto de propiedades, atributos ó datos) de un objeto sólo debe ser modificado por medio de los métodos del propio objeto.

La técnica de encapsulación, es conocida como ocultación de datos, le permite que los atributos de un objeto pueden ocultarse (superficialmente) para que no sean accedidos desde fuera de la definición de una clase. Para ello, es necesario nombrar los atributos con un prefijo de doble subrayado: `__atributo`.

```
>>> class Factura:
...     __tasa = 19
...     def __init__(self, unidad, precio):
...         self.unidad = unidad
...         self.precio = precio
...     def por_pagar(self):
...         total = self.unidad * self.precio
...         impuesto = total * Factura.__tasa / 100
...         return(total + impuesto)
...
>>> compra1 = Factura(12, 110)
>>> print compra1.unidad
12
>>> print compra1.precio
110
>>> print compra1.por_pagar(), "bitcoins"
(1570, 'bitcoins')
>>> print Factura.__tasa
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: class Factura has no attribute '__tasa'
>>>
```

Python protege estos atributos cambiando su nombre internamente. A sus nombres agrega el nombre de la clase:

```
objeto._NombreClase__NombreAtributo
```

```
>>> print compra1._Factura__tasa
19
>>>
```

Por hacer: TODO terminar de escribir esta sección

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

9.6 Polimorfismo

La técnica de polimorfismo de la *POO* (página 194) significa la capacidad de tomar más de una forma. Una operación puede presentar diferentes comportamientos en diferentes instancias. El comportamiento depende de los tipos de datos utilizados en la operación. El polimorfismo es ampliamente utilizado en la aplicación de la herencia.

Por hacer: TODO escribir esta sección

9.6.1 Sobrecarga de métodos

La *sobrecarga de métodos* es también es conocida por *Overriding Methods*, le permite sustituir un método proveniente de la Clase Base, en la Clase Derivada debe definir un método con la **misma forma** (es decir, mismo nombre de método y mismo número de parámetros que como está definido en la Clase Base).

```
>>> class Persona():
...     def __init__(self):
...         self.cedula = 13765890
...     def mensaje(self):
...         print("mensaje desde la clase Persona")
...
>>> class Obrero(Persona):
...     def __init__(self):
...         self.__especialista = 1
...     def mensaje(self):
...         print("mensaje desde la clase Obrero")
...
>>> obrero_planta = Obrero()
>>> obrero_planta.mensaje()
mensaje desde la clase Obrero
>>>
```

Lo que se logra definiendo el método `mensaje()` en la Clase Derivada (`Obrero`) se conoce como **Método Overriding** (cuando se cree el objeto (en este caso `obrero_planta` y se llame al método `mensaje()`, este será tomado de la propia clase y no de la Clase Base `Persona`). Si **comenta o borra** el método `mensaje()` de la clase `Obrero` (Clase Derivada) y corre nuevamente el código, el método llamado será el `mensaje()` de la Clase Base `Persona`.

9.6.2 Sobrecarga de Operadores

La *sobrecarga de operadores* es también es conocida por *Overloading Operators*, trata básicamente de lo mismo que la **sobrecarga de métodos** pero pertenece en esencia al ámbito de los operadores aritméticos, binarios, de comparación y lógicos.

```
>>> class Punto:
...     def __init__(self,x = 0,y = 0):
...         self.x = x
...         self.y = y
...     def __add__(self,other):
...         x = self.x + other.x
...         y = self.y + other.y
...         return x, y
...
>>> punto1 = Punto(4,6)
>>> punto2 = Punto(1,-2)
>>> print punto1 + punto2
(5, 4)
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

9.7 Objetos de tipos integrados

Existe otros *tipos de datos integrados* (página 25) en el interprete Python, que para muchos no son de uso frecuente, los cuales se describen a continuación:

9.7.1 Ellipsis

Este tipo tiene un solo valor. Hay un solo objeto con este valor. Se accede a este objeto `elipsis` a través del nombre incorporado «`Ellipsis`». Se utiliza para indicar la presencia de la sintaxis «`...`» en una porción o la notación de corte extendida. Su valor de verdad es `True`.

```
>>> type(Ellipsis)
<type 'ellipsis'>
```

9.7.2 None

Este tipo tiene un solo valor. Hay un solo objeto con este valor. Se accede a este objeto a través del nombre incorporado «`None`». Se utiliza para indicar la ausencia de un valor en muchas situaciones, por ejemplo, se devuelve desde las funciones que no devuelven nada explícitamente. Su valor de verdad es `False`.

```
>>> type(None)
<type 'NoneType'>
```

9.7.3 NotImplemented

Este tipo tiene un solo valor. Hay un solo objeto con este valor. Se accede a este objeto a través del nombre incorporado «`NotImplemented`». Los métodos numéricos y los métodos de comparación enriquecidos (como `__eq__()`, `__lt__()` y amigos), para indicar que la comparación no se implementa con respecto al otro tipo, es decir, pueden devolver este valor si no implementan la operación para los operandos proporcionados. (El intérprete luego intentará la operación reflejada, o algún otro respaldo «`fallback`», dependiendo del operador). Su valor de verdad es `True`.

```
>>> type(NotImplemented)
<type 'NotImplementedType'>
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

9.8 Clases de tipos integrados

Python integra varias clases de tipos en el modulo `__builtin__`, a continuación se describen algunas clases:

9.8.1 Clases generales

Las clases de uso general se describen a continuación:

buffer

El objeto de la clase `buffer()` crea un nuevo objeto de búfer que haga referencia al objeto dado. El búfer hará referencia a una porción del objeto de destino desde el inicio del objeto (o en el desplazamiento especificado). La división se extenderá hasta el final del objeto de destino (o con el tamaño especificado).

```
>>> cadena = 'Hola mundo'
>>> cadena
'Hola mundo'
>>> cadena[5:10]
```

(continué en la próxima página)

(proviene de la página anterior)

```
'mundo'
>>> type(cadena)
<type 'str'>
>>> cadena_buffer = buffer(cadena, 5, 5)
>>> type(cadena_buffer)
<type 'buffer'>
>>> cadena_buffer
<read-only buffer for 0x7f42121d3810, size 5, offset 5 at 0x7f42121d23b0>
>>> print cadena_buffer
mundo
```

El búfer en este caso anterior es una sub-cadena, inicia en la posición 5 con un ancho de 5 caracteres y es no toma espacio de almacenamiento extra - eso referencia a `cortar` una cadena de caracteres.

Este ejemplo anterior no es muy útil para cadenas de caracteres cortas como esta, pero eso puede ser necesario cuando usa un gran numero de data. Este ejemplo puede usar un tipo mutable `bytearray()`:

```
>>> cadena = bytearray(1000000)
>>> type(cadena)
<type 'bytearray'>
>>> cadena_buffer = buffer(cadena, 1)
>>> cadena_buffer
<read-only buffer for 0x7f42121d3870, size -1, offset 1 at 0x7f42121d2270>
>>> type(cadena_buffer)
<type 'buffer'>
>>> cadena_buffer[0]
'\x00'
>>> cadena[1]
0
>>> cadena[1] = 5
>>> cadena[1]
5
>>> cadena_buffer[0]
'\x05'
```

Esto puede ser muy útil si usted quiere tener más que una vista en la data y no quiere (o puede) contener múltiples copias en memoria.

Note que el búfer ha sido remplazado por un mejor método llamado *memoryview()* (página 222) en Python 3, aunque se puede usar en Python 2.7.

Note también que usted no puede implementar una interfaz búfer para sus propios objetos sin profundizando en la API de C, ej. usted no puede hacer eso con puramente con código Python.

En general un `slice` tomará extra almacenamiento, entonces, si `cadena[5:10]` será una copia. Si usted define `cadena_buffer = cadena[5:10]` y entonces `del cadena`, eso liberaría la memoria que fue tomada por `cadena`, proveyendo que `cadena_buffer` fue copiada. (Para usar esto usted necesita una gran cadena de caracteres, en este ejemplo `cadena` y rastrear el uso de la memoria de Python). Es sin embargo mucho más eficiente que hacer la copia si no existe mucha data involucrada.

bytes

El objeto de la clase `bytes` es agregada en Python 2.6 como un sinónimo para el tipo *str* (página 46) y este también soporta la notación `b''`.

El uso principal de bytes en Python 2.6 será escribir pruebas de tipo de objeto como `isinstance(x, bytes)`. Esto ayudará al convertidor `2to3`, que no puede decir si el código 2.x pretende que las cadenas contengan caracteres o bytes de 8 bits; ahora puede usar `bytes` o `str` para representar exactamente su intención, y el código resultante también será correcto en Python 3.0.

```
>>> arreglo = bytes("Python es interesante.")
>>> print arreglo
Python es interesante.
>>> type(arreglo)
<type 'str'>
```

quit

Es el método constructor de la clase `Quitter` incluida en el módulo `site` el cual le permite salir de la consola interactiva Python:

```
>>> quit
Use quit() or Ctrl-D (i.e. EOF) to exit
>>> quit()
$
```

De esta forma puede salir de la consola interactiva Python y volviendo al interprete del Shell de comando.

slice

La clase `slice` crea un objecto `slice`, esto es usado por el extendido `slicing` por ejemplo:

```
>>> a = range(20)
>>> a[0:10:2]
[0, 2, 4, 6, 8]
```

La sintaxis es la siguiente:

```
>>> slice(stop)
>>> slice(start, stop[, step])
```

Por hacer: TODO escribir sobre esta clase integrada.

---

staticmethod

Los métodos estáticos en Python son extremadamente similar a los métodos de nivel clase en python, la diferencia esta que un método estático es enlazado a una clase más bien que los objectos para esa clase.

Esto significa que un método estático puede ser llamado sin un objeto para esa clase. Esto también significa que los métodos estáticos no pueden modificar el estado de un objeto como ellos no pueden enlazarse a ese.

Los métodos estáticos Python puede crearse en dos formas, usando el aprovechamiento `staticmethod()` o el decorador `@staticmethod`:

La clase `staticmethod()` convierte una función a un método estático. Un método estático no recibe un primer argumento implícito. La sintaxis es la siguiente:

```
>>> staticmethod(function) -> método
```

Para declarar un método estático, a continuación vea el siguiente ejemplo:

```
>>> class Calculador:
...     def sumaNumeros(x, y):
...         return x + y
...     # crea un static method sumaNumeros
...     sumaNumeros = staticmethod(sumaNumeros)
```

(continué en la próxima página)

(proviene de la página anterior)

```
...
>>> print 'Resultado:', Calculador.sumaNumeros(15, 110)
Resultado: 125
>>> print 'Resultado:', Calculador().sumaNumeros(15, 110)
Resultado: 125
```

En el ejemplo anterior usted puede notar que se llamo al método `sumaNumeros` sin crear un objeto. Se puede llamar en la clase (por ejemplo, `Clase.funcion()`) o en una instancia (por ejemplo, `Clase().funcion()`). La instancia se ignora a excepción de su clase.

Los métodos estáticos son similares a los métodos estáticos `Java` o `C++`. Para un concepto más avanzado, mire la clase *classmethod* (página 221) integrada en el interprete.

La clase `staticmethod` introduce un cambio en la versión 2.4, agregando sintaxis de *decorador* (página 231) de función. La sintaxis es la siguiente:

```
class Clase:
    @staticmethod
    def funcion(argumento1, argumento2, ...):
        ...
```

Un ejemplo del uso de *decoradores* (página 231) para `staticmethod` a continuación:

```
>>> class Calculador:
...     @staticmethod
...     def sumaNumeros(x, y):
...         return x + y
...
>>> print 'Resultado:', Calculador.sumaNumeros(15, 110)
Resultado: 125
```

Este código fuente es enteramente idéntico al primer ejemplo (usando `@staticmethod`), solo que no usa la agradable sintaxis de *decorador* (página 231).

Finalmente, se usa el método `staticmethod()` escasamente. Hay muchas situaciones donde los métodos estáticos son necesarios en Python.

9.8.2 Clases de secuencias

Las clases de tipos secuencias se describen a continuación:

enumerate

La clase `enumerate` devuelve un objeto *enumerate*. El iterable debe ser otro objeto que soporte *iteradores* (página 92). El objeto *enumerate* produce pares que contiene una cuenta (desde donde inicia, el cual el valor por defecto es cero) y un valor producido por el argumento iterable.

Cuando la iteración de la secuencia llega al final se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración. El objeto enumerate es muy útil para obtener una lista indexada como: `(0, seq[0]), (1, seq[1]), (2, seq[2]), ...`.

```
>>> enumerar = enumerate(xrange(3))
>>> enumerar.next()
(0, 0)
>>> enumerar.next()
(1, 1)
>>> enumerar.next()
(2, 2)
>>> enumerar.next()
```

(continué en la próxima página)

(proviene de la página anterior)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior usa una secuencia numérica de 3 elementos generada por la función integrada *xrange()* (página 121).

A continuación se le pasa el parámetro de *inicio* con el valor *1* de la secuencia generada por la clase `enumerate`:

```
>>> enumerar = enumerate(xrange(3), 1)
>>> enumerar.next()
(1, 0)
>>> enumerar.next()
(2, 1)
>>> enumerar.next()
(3, 2)
>>> enumerar.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior usa una secuencia numérica de 3 elementos generada con el valor inicial de *1* por la función integrada *xrange()* (página 121).

reversed

La clase `reversed` devolver un *iterador* (página 92) inverso sobre los valores de la secuencia, cuando la iteración de la secuencia llega al final se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

```
>>> inversa = reversed(xrange(3))
>>> inversa.next()
2
>>> inversa.next()
1
>>> inversa.next()
0
>>> inversa.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

En el ejemplo anterior usa una secuencia numérica de 3 elementos generada por la función integrada *xrange()* (página 121).

9.8.3 Clases de archivos

Las clases de tipos *archivos* se describen a continuación:

file()

El objeto `file()` se implementan con el paquete del lenguaje C `stdio` y se pueden crear con la función interna *open()* (página 119). También son el resultado de otras funciones y métodos internos, por ejemplo, `os.popen()` y `os.fdopen()` y el método `makefile()` de los objetos `socket`.

Cuando falla una operación de archivos por una cuestión de E/S, se lanza la excepción *IOError* (página 191). Esto incluye situaciones donde la operación no esté definida por cualquier motivo, como usar *seek()* (página 217) en un dispositivo `tty` o intentar escribir en un archivo abierto para lectura.

Métodos

El objeto `file()` implementa los siguientes métodos integrados:

close()

El método `close()` permite cerrar la manipulación del archivo. No es posible escribir ni leer en un archivo cerrado. Cualquier operación que requiera que el archivo esté abierto lanzará *IOError* (página 191) si el archivo se ha cerrado. Está permitido llamar a `close()` más de una vez.

Una vez que se terminó de usar el archivo es necesario cerrarlo, para liberar los recursos tomados por el manejo del archivo. Eso se hace con la sentencia `archivo.close()`:

```
>>> archivo.close() # cierra el archivo datos.txt
```

Luego de lo cual no se puede acceder al archivo `datos.txt`, si intenta una llamada a la método *archivo.read()* (página 216) devuelve una excepción *ValueError* (página 193), porque el archivo está cerrado:

```
>>> archivo.close()
>>> archivo.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file
```

Truco: Para más detalles: https://docs.python.org/2.7/tutorial/inputoutput.html

flush()

El método `flush()` permite descargar el bufér interno, como la función de lenguaje C `fflush()` de la librería `stdio`. Puede no tener efecto en ciertos objetos similares a los archivos.

Python automáticamente flushes los archivos cuando son cerrados. Pero usted podría to flush la data antes de cerrar cualquier archivo.

```
>>> archivo = open("datos.txt", "wb") # Abre un archivo
>>> print "Nombre del archivo: ", archivo.name
Nombre del archivo:  datos.txt
>>> archivo.flush()
... # Aquí eso no hace nada, pero usted puede
... # llamarlo con la operación read.
>>> archivo.close() # Cerrar archivo abierto
```

isatty()

El método `isatty()` devuelve `True` si el archivo está conectado a un dispositivo `tty` (un terminal interactivo de líneas de orden), en caso contrario, `False`.

Nota: Si un objeto similar a los archivos no está asociado a un archivo real, no debe implementar este método.

```
>>> archivo = open('datos.txt', 'r')
>>> archivo.isatty()
False
```

fileno()

El método `fileno()` devuelve el «descriptor de archivo» utilizado por la implementación subyacente para solicitar operaciones E/S del sistema operativo. Puede ser útil para interfaces de bajo nivel que utilicen descriptores de archivos, por ejemplo, el módulo `fcntl` o `os.read()` y similares.

Nota: Si un objeto similar a los archivos no tiene un descriptor de archivo, no debe implementar este método.

```
>>> archivo = open("datos.txt",mode="r")
>>> archivo.fileno()
6
```

next()

El método `next()` permite usar un iterador para tratar cada linea del archivo como el próximo valor, cuando la iteración del archivo llega al final se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

```
>>> archivo = open('/etc/hostname')
>>> archivo
<open file '/etc/hostname', mode 'r' at 0x7fa44ba379c0>
>>> archivo.__iter__()
<open file '/etc/hostname', mode 'r' at 0x7fa44ba379c0>
>>> iter(archivo)
<open file '/etc/hostname', mode 'r' at 0x7fa44ba379c0>
>>> archivo is archivo.__iter__()
True
>>> linea = archivo.__iter__()
>>> linea.next()
'laptop\n'
>>> linea.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

read()

El método `read()` permite leer el contenido del archivo. El argumento es opcional y si no se especifica (o es -1) devuelve el contenido de todo el archivo. Una vez que se leyó todo el archivo, una nueva llamada a la función devuelve una cadena vacía (“”).

```
>>> archivo = open('datos.txt', 'r')
>>> archivo.read()
'Este es una prueba \ny otra prueba'
>>> archivo.read()
''
```

Si desea recibir una salida formateada por consola leyendo un archivo, a continuación un ejemplo:

```
>>> archivo = open('datos.txt', 'r')
>>> contenido = archivo.read()
>>> print contenido
Este es una prueba
y otra prueba
```

readline()

El método `readline()` permite leer una sola línea del archivo, devuelve al final de la línea el carácter de nueva línea y solo se omite en la última línea del archivo (si no termina con el carácter de nueva línea). Esto hace que el valor de retorno no sea ambiguo. Si devuelve una cadena de caracteres vacía se alcanzó el fin del archivo, mientras que una línea en blanco se representa con un carácter de nueva línea.

```
>>> archivo = open('datos.txt', 'r')
>>> print archivo.readline() # lee la linea "Este es una prueba "
>>> print archivo.readline() # lee la linea "y otra prueba"
>>> print archivo.readline()

>>>
```

readlines()

El método `readlines()` devuelve una lista que contiene todas las líneas del archivo.

```
>>> archivo = open('datos.txt', 'r')
>>> lineas = archivo.readlines()
>>> print lineas
['Este es una prueba \n', 'y otra prueba']
```

seek()

El método `seek()` mueve la posición actual del cursos del archivo, como la función del lenguaje C `fseek()` de la librería `stdio`. No devuelve ningún valor.

El método `seek()` lleva la siguiente nomenclatura:

```
>>> seek(posicion_actual[, punto_referencia])
```

A continuación, un ejemplo que escribir y leer el archivo `datos.txt` agregando una lista de lineas al principio del archivo, como al final del archivo:

```
>>> archivo = open('datos.txt', 'w')
>>> lista_de_lineas = ["Esta es la 1er linea", \
...     "Esta es la 2da linea", "Esta es la 3era linea"]
>>> archivo.writelines("\n".join(lista_de_lineas))
>>> archivo.close()
>>> archivo = open('datos.txt', 'r')
>>> archivo.next()
'Esta es la 1er linea\n'
>>> archivo.seek(8)
>>> archivo.next()
'la 1er linea\n'
>>> archivo.next()
'Esta es la 2da linea\n'
>>> archivo.next()
'Esta es la 3era linea'
>>> archivo.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> archivo.close()
```

En el ejemplo anterior, puede ver que se escriben tres lineas y se pasa como argumento `posicion_actual` el valor *8* el cual posiciona el curso de búsqueda en dicha posición de la primera linea con `archivo.seek(8)` y muestra una parte de la linea.

El argumento `punto_referencia` es opcional, con un valor predeterminado de `0` (es el principio del archivo); otros valores posibles son `1` (la posición actual del archivo) y `2` (el final del archivo). No hay valor de retorno.

```
>>> archivo = open('datos.txt', 'w')
>>> lista_de_lineas = ["Esta es la 1er linea", \
...     "Esta es la 2da linea", "Esta es la 3era linea"]
>>> archivo.writelines("\n".join(lista_de_lineas))
>>> archivo.close()
>>> archivo = open('datos.txt', 'r')
>>> archivo.next()
'Esta es la 1er linea\n'
>>> archivo.seek(8)
>>> archivo.next()
'la 1er linea\n'
>>> archivo.close()
>>> archivo = open('datos.txt', 'rw+')
>>> nuevas_lineas = ["\nEsta es la 4ta linea", \
...     "Esta es la 5ta linea"]
>>> # Escribe la secuencia de la lineas al final del archivo.
... archivo.seek(0, 2)
>>> archivo.writelines("\n".join(nuevas_lineas))
>>> # Ahora lea completamente el archivo desde el inicio.
... archivo.seek(0,0)
>>> for elemento in range(1, 6):
...    linea = archivo.next()
...    print "Linea No %d - %s" % (elemento, linea)
...
Linea No 1 - Esta es la 1er linea

Linea No 2 - Esta es la 2da linea

Linea No 3 - Esta es la 3era linea

Linea No 4 - Esta es la 4ta linea

Linea No 5 - Esta es la 5ta linea
>>> # Cerrar archivo abierto
... archivo.close()
>>>
```

En el ejemplo anterior se pudo usar el método `seek()` con el argumento `punto_referencia` al final del archivo para agregar nuevas lineas y luego se uso de nuevo el argumento `punto_referencia` para ubicarse al inicio del archivo para mostrar todo el contenido del archivo.

tell()

El método `tell()` devuelve la posición actual del archivo, como la función del lenguaje C `ftell()` de la librería `stdio`.

```
>>> archivo = open('/etc/hostname')
>>> archivo.tell()
0
>>> linea = iter(archivo)
>>> linea.next()
'debacagua9\n'
>>> archivo.tell()
11
>>> len('debacagua9\n')
11
>>> linea.next()
Traceback (most recent call last):
```

(continué en la próxima página)

(proviene de la página anterior)

```
  File "<stdin>", line 1, in <module>
StopIteration
>>> archivo.tell()
11
```

Cuando la iteración de la secuencia llega al final se llama a la excepción *StopIteration* (página 192) y se causa el detener la iteración.

truncate()

```
>>> archivo = open('datos.txt', 'w')
>>> archivo.write('Este es una prueba \ny otra prueba')
>>> archivo.truncate(20)
>>> archivo.close()
>>> archivo = open('datos.txt', 'r')
>>> archivo.read()
'Este es una prueba \n'
```

El método `truncate()` trunca el archivo. Si se proporciona el argumento opcional, el archivo se trunca a (como mucho) ese tamaño. El tamaño depende de la posición actual. La disponibilidad de esta función depende de la versión del sistema operativo (por ejemplo, no todas las versiones de Unix dan soporte a esta operación).

write()

El método `write()` permite escribir el contenido de la cadena de texto al archivo, y devuelve la cantidad de caracteres escritos.

Para escribir algo que no sea una cadena de caracteres, antes se debe convertir a cadena de caracteres.

```
>>> archivo = open('datos.txt', 'w')
>>> # escribe el archivo datos.txt
... archivo.write('Este es una prueba \ny otra prueba')
>>>
```

writelines()

El método `writelines()` escribe una lista de cadenas al archivo. No se devuelve ningún valor. El nombre es paralelo a `readlines()`, `writelines()` no añade separadores de línea.

```
>>> archivo = open('datos.txt', 'w')
>>> lista_de_lineas = ['Plone es el más poderoso, ', \
...     'escalable, seguro ', 'y longevo CMS, ', \
...     'escrito en Python.']
>>> archivo.writelines("\n".join(lista_de_lineas))
>>> archivo.close()
```

Atributos

Los objetos archivo también ofrecen otros atributos interesantes. No son necesarios para los objetos de interfaz tipo archivo, pero deberían implementarse si tienen sentido en un objeto particular.

closed

El atributo `closed` del objeto *file* (página 214) de tipo *booleano* (página 44) indica el estado actual. Es un atributo de sólo lectura, que se cambia mediante el método *close()* (página 215). Puede no estar disponible en todos los objetos con interfaz tipo archivo.

```
>>> archivo = open('datos.txt', 'w')
>>> archivo.closed
False
>>> archivo.close()
>>> archivo.closed
True
```

mode

El atributo `mode` del objeto *file* (página 214), es el modo de E/S del archivo. Si se creó el archivo con la función integrada *open()* (página 119), será el valor del parámetro `mode`. Es un atributo de sólo lectura y puede no estar disponible en todos los objetos con interfaz tipo archivo.

```
>>> archivo = open('datos.txt', 'w')
>>> archivo.mode
'w'
```

name

El atributo `name` del objeto *file* (página 214), es el nombre del archivo si se creó el objeto archivo mediante la función integrada *open()* (página 119), el nombre del archivo. En caso contrario, alguna cadena que indique el origen del archivo, de la forma «<...>». Es un atributo de sólo lectura y puede no estar disponible en todos los objetos con interfaz tipo archivo.

```
>>> archivo = open('datos.txt', 'w')
>>> archivo.name
'datos.txt'
```

encoding

El atributo `encoding` del objeto *file* (página 214), es el encoding del archivo.

```
>>> with open("datos.txt",mode="r") as archivo:
...     print "Encoding por defecto:", archivo.encoding
...     archivo.close()
...
Encoding por defecto: None
```

softspace

El atributo `softspace` del objeto *file* (página 214) del tipo *booleano* (página 44) indica si se debe escribir un espacio antes de escribir otro valor al usar la sentencia *print* (página 153). Las clases que intenten simular un objeto archivo deberían tener un atributo escribible `softspace`, que debería inicializarse a cero.

Esto será automático en la mayoría de las clases implementadas en Python (se debe tener cuidado en las clases que redefinan el acceso a los atributos). Los tipos implementados en el lenguaje C tendrán que proporcionar un atributo `softspace` escribible.

Nota: Este atributo no se usa para controlar la sentencia `print`, sino para permitir que la implementación de `print` lleve la cuenta de su estado interno.

```
>>>
>>> archivo = open('datos.txt', 'w')
>>> archivo.softspace
0
```

9.8.4 Clases de objetos

Las clases de objetos se describen a continuación:

classmethod

La clase `classmethod` convierte una función para ser un método de clase. Un método de clase recibe la clase como primer argumento implícito, al igual que un método de instancia recibe la instancia. La sintaxis es la siguiente:

```
>>> classmethod(function) -> método
```

Para declarar un método de clase, a continuación vea el siguiente ejemplo:

```
>>> def sumaNumeros(cls, x, y):
...     return x + y
...
>>> type(sumaNumeros)
<type 'function'>
>>> class Calculador:
...     # crea un static method sumaNumeros
...     sumaNumeros = classmethod(sumaNumeros)
...
>>> Calculador.sumaNumeros(15, 110)
125
>>> Calculador().sumaNumeros(15, 110)
125
>>> type(Calculador.sumaNumeros)
<type 'instancemethod'>
```

La clase `classmethod` introduce un cambio en la versión 2.4, agregando sintaxis de *decorador* (página 231) de función. La sintaxis es la siguiente:

```
class Clase:
    @classmethod
    def funcion(cls, argumento1, argumento2, ...):
        ...
```

Un ejemplo del uso de *decoradores* (página 231) para `classmethod` a continuación:

```
>>> class Clase:
...     @classmethod
...     def funcion(cls, argumento1, argumento2):
...         return argumento1 + argumento2
...
>>> Clase.funcion(2, 3)
5
>>> Clase().funcion(2, 3)
5
```

Se puede llamar en la clase (por ejemplo, `Clase.funcion()`) o en una instancia (por ejemplo, `Clase().funcion()`). La instancia se ignora a excepción de su clase. Si se llama a un método de clase para una clase derivada, el objeto de clase derivada se pasa como el primer argumento implícito.

Los métodos de clase son diferentes a los métodos estáticos `C++` o `Java`. Si quieres eso, mira la clase *staticmethod* (página 212) integrada en el interprete.

Por hacer: TODO terminar de escribir sobre la clase integrada classmethod.

---

memoryview

La clase `memoryview` devuelve un objeto *vista de memoria* del argumento dado.

Antes de introducir a que son las *vistas de memoria*, necesita entender primero sobre del *protocolo Búfer* de Python.

¿Qué es protocolo Búfer?

Este protocolo provee una forma de acceder la data interna de un objeto. Esta data interna es un arreglo de memoria o un búfer. El *protocolo Búfer* le permite un objeto para exponer esa data interna (búfers) y el otro para acceder a esos búfers sin tener que copiar intermediamente.

Este protocolo es solamente accesible al usar el nivel API de C y no usando el normal código base. Por lo tanto, para exponer el mismo protocolo a la base de código Python normal, las vistas de memoria están presentes.

¿Qué es una vista de memoria?

La vista de memoria es una forma segura de exponer el protocolo búfer en Python. Eso le permite a usted acceder a los búfers internos de un objeto para creación de un objeto de vista de memoria.

¿Por que el protocolo búfer y las vistas de memoria son importantes?

Necesita recordar que cada vez que ejecuta alguna acción en un objeto (llamar a una función de un objeto, cortar un arreglo), Python necesita crear una copia del objeto.

Si usted tiene una gran data para trabajar con ella (ej. data binaria de una imagen), debería crear innecesariamente copias de enormes trozos de datos, que casi no sirve de nada.

Usando el *protocolo búfer*, puede dar otros accesos al objeto para usar/modificar data grande sin realizar copias de eso. Esto hace que el programa use menos memoria y incremente la velocidad de ejecución.

¿Como exponer el protocolo búfer usando las vistas de memoria?

Los objetos de *vista de memoria* son creados usando la sintaxis:

```
>>> memoryview(objecto)
```

El método constructor `memoryview()` toma un simple parámetro:

`objecto` - es el objeto dado como parámetro el cual su data interna es expuesta.

`objecto` debe ser un tipo el cual soportar el *protocolo búfer* (`bytes`, `bytearray`). Devuelve el valor de un objeto de vista de memoria del objeto dado como parámetro desde el método `memoryview()`.

A continuación, un ejemplo donde se crea una *vista de memoria* usando el tipo `bytearray` previamente creado:

```
>>> cadena = bytearray(1000000)
>>> memoryview(cadena)
<memory at 0x7f6202179cc8>
>>> memoryview(cadena).format
'B'
>>> memoryview(cadena).itemsize
1L
>>> memoryview(cadena).ndim
```

(continué en la próxima página)

(proviene de la página anterior)

```
1L
>>> memoryview(cadena).readonly
False
>>> memoryview(cadena).shape
(1000000L,)
>>> memoryview(cadena).strides
(1L,)
>>> memoryview(cadena).suboffsets
```

En el ejemplo anterior se crea una *vista de memoria* de un tipo `bytearray` mostrando los diversos atributos disponibles.

Continuando el ejemplo anterior, se crea una *vista de memoria* de un tipo *buffer* (página 210) usando el objeto `cadena` previamente creado:

```
>>> cadena_buffer = buffer(cadena, 1)
>>> memoryview(cadena_buffer)
<memory at 0x7f6202179cc8>
>>> memoryview(cadena_buffer).format
'B'
>>> memoryview(cadena_buffer).itemsize
1L
>>> memoryview(cadena_buffer).ndim
1L
>>> memoryview(cadena_buffer).readonly
True
>>> memoryview(cadena_buffer).shape
(999999L,)
>>> memoryview(cadena_buffer).strides
(1L,)
>>> memoryview(cadena_buffer).suboffsets
```

En el ejemplo anterior se crea una *vista de memoria* de un tipo *buffer* (página 210) mostrando los diversos atributos disponibles.

A continuación, otro ejemplo donde se crea una *vista de memoria* usando el objeto `bytearray` previamente creado:

```
>>> randomBA = bytearray('ABC', 'utf-8')
>>> randomBA
bytearray(b'ABC')
>>> vm = memoryview(randomBA)
>>> vm
<memory at 0x7fafc7136c30>
>>> print vm[0]
A
>>> print vm[1]
B
>>> print vm[2]
C
```

Continuando el ejemplo anterior, se puede crear una *lista* (página 58) desde una *vista de memoria* usando el objeto `vm` previamente creado:

```
>>> list = []
>>> for item in range(3): list.append(vm[item])
...
>>> list
['A', 'B', 'C']
```

Continuando el ejemplo anterior, se puede crear *cadena de caracteres* (página 46) desde una *vista de memoria* usando el objeto `vm` previamente creado:

```
>>> cad = ""
>>> for item in range(3): cad += vm[item]
...
>>> print cad
ABC
```

Aquí, es creada un objeto *vista de memoria* llamado `vm` desde un objeto `bytearray` llamado `randomBA`.

Entonces, es accedido al índice 0 posición `vm` "A" y el valor es impreso. Luego, es accedido al índice 1 posición `vm` "B" y el valor es impreso. También, es accedido al índice 2 posición `vm` "C" y el valor es impreso.

Finalmente, es accedido todos los índices del objeto `vm` y convertidos a una lista.

A continuación, otro ejemplo donde se modifica la data interna usando vista de memoria:

```
>>> randomBA = bytearray('ABC', 'utf-8')
>>> print 'Antes de actualizar:', randomBA
Antes de actualizar: ABC
>>> vm = memoryview(randomBA)
>>> chr(90)
'Z'
>>> vm[1] = chr(90)
>>> print 'Después de actualizar:', randomBA
Después de actualizar: AZC
```

Aquí, se actualiza el indice 1 de la *vista de memoria* a un valor ASCII - 90 (Z) usando la función *chr()* (página 130). Desde, el objeto de *vista de memoria* `vm` referencia al mismo búfer/memoria, actualiza el índice en el `vm` también actualiza el `randomBA`.

Desde adentro internamente el tipo `bytearray` almacena valores `ASCII` para el alfabeto, es decir, cada posición de la lista se debe indicar con su equivalente numérico en la tabla `ASCII`.

```
>>> chr(65)
'A'
>>> chr(66)
'B'
>>> chr(67)
'C'
>>> chr(90)
'Z'
```

Entonces se usa la función *chr()* (página 130) para indicar su equivalente en la tabla de valores `ASCII`.

object

El objeto de la clase `object` es el tipo más básico de objeto, es integrado en el módulo `__builtin__`. Este objeto se usa como *herencia* (página 202) cuando se crea una nueva clase en Python.

Todo, incluyendo las clases y tipos de Python son instancias de `object`. Para corroborar si un objeto es instancia de una clase se utiliza la función *isinstance()* (página 142).

```
>>> object
<type 'object'>
```

property

La clase `property` típicamente es usado para definir un atributo property. La sintaxis es la siguiente:

```
>>> property(fget=None, fset=None,
...     fdel=None, doc=None) # devuelve atributo property
```

El parámetro `fget` es una función a ser usada para obtener un valor de un atributo, y igualmente el parámetro `fset` es una función para definir el valor de un atributo, y el parámetro `fdel` es una función para eliminar un atributo.

El método `property()` devuelve un atributo `property` donde es dado el método `getter`, `setter` y `deleter`.

Si no hay argumentos son dados, el método `property()` devuelven un atributo base `property` que no contienen ningún `getter`, `setter` o `deleter`. Si `doc` no es proveído, método `property()` toma el *docstring* (página 50) de la función `getter`.

A continuación, un ejemplo sencillo:

```
>>> class Persona:
...     def __init__(self, nombre):
...         self._nombre = nombre
...
...     def getNombre(self):
...         print 'Obteniendo nombre'
...         return self._nombre
...
...     def setNombre(self, valor):
...         print 'Definiendo nombre a ' + valor
...         self._nombre = valor
...
...     def delNombre(self):
...         print 'Eliminando nombre'
...         del self._nombre
...
...     # Define la property para usar los métodos getNombre,
...     # setNombre y delNombre
...     nombre = property(getNombre, setNombre, delNombre, 'Atributo property␣
↪nombre')
...
>>> persona1 = Persona('Leo')
>>> print persona1.nombre
Obteniendo nombre
Leo
>>> persona1.nombre = 'Leonardo'
>>> print persona1.nombre
Leonardo
>>> dir(persona1)
['__doc__', '__init__', '__module__', '_nombre', 'delNombre',
'getNombre', 'nombre', 'setNombre']
>>> persona1.delNombre()
Eliminando nombre
>>> dir(persona1)
['__doc__', '__init__', '__module__', 'delNombre', 'getNombre',
'nombre', 'setNombre']
>>> print persona1.nombre
Leonardo
>>> del persona1.nombre
>>> print persona1.nombre
Obteniendo nombre
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in getNombre
AttributeError: Persona instance has no attribute '_nombre'
```

Cuando se elimina `persona1.delNombre()` puede notar que `_nombre` ya no esta disponible y si se vuelve a imprimir el valor de nombre `print persona1.nombre` aun muestra el valor inicializado con el método `setNombre`, entonces al ejecutar `del persona1.nombre` se elimina por completo el valor en memoria, luego si intenta mostrar el valor del *atributo property* `nombre` lanza *AttributeError* (página 191) por no encontró `_nombre` el cual es usado como la variable privado para almacenar el nombre de una Persona.

Se definió lo siguiente:

- Un método `getter` getNombre() para obtener el nombre de la persona,
- Un método `setter` setNombre() para definir el nombre de la persona,
- Un método `deleter` delNombre() para eliminar el nombre de la persona.

Ahora tiene definido un *atributo property* `nombre` llamando al método `property()`.

Como se mostró en el código anterior, la referencia `persona1.nombre` internamente llama al método `getName()` como *getter*, `setName()` como *setter* y `delName()` como *deleter* a través de las salidas impresas presente dentro de los métodos.

También se definió el *docstring* (página 50) del atributo con el valor *"Atributo property nombre"*.

Otra alternativa son los *decoradores* (página 231) facilitan la definición de nuevas propiedades o la modificación de las existentes:

A continuación se creará un *atributo property* con métodos `getter`, `setter` y `deleter` usando el decorador `@property` en vez de usar el método `property()`, usted puede usar el decorador Python `@property` para asignar el método `getter`, `setter` y `deleter`:

```
>>> class Persona:
...     def __init__(self, nombre):
...         self._nombre = nombre
...
...     @property
...     def nombre(self):
...         print 'Obteniendo nombre'
...         return self._nombre
...
...     @nombre.setter
...     def nombre(self, valor):
...         print 'Definiendo nombre a ' + valor
...         self._nombre = valor
...
...     @nombre.deleter
...     def nombre(self):
...         print 'Eliminando nombre'
...         del self._nombre
...
>>> persona1 = Persona('Leo')
>>> print 'El nombre es:', persona1.nombre
El nombre es: Obteniendo nombre
Leo
>>> persona1.nombre = 'Leonardo'
>>> print persona1.nombre
Leonardo
>>> dir(persona1)
['__doc__', '__init__', '__module__', '_nombre', 'nombre']
>>> del persona1.nombre
>>> dir(persona1)
['__doc__', '__init__', '__module__', '_nombre', 'nombre']
>>> print persona1.nombre
Obteniendo nombre
Leo
```

Aquí, en vez de usar el método `property()`, es usado el *decorador* (página 231) @property.

Primero especifica que el método `nombre()` es un atributo de la clase `Persona`. Esto es hecho usando la sintaxis `@property` antes el método *getter* como se muestra en el código anterior.

Seguidamente se usa el nombre del atributo `nombre` para especificar los métodos *setter* y *deleter*.

Esto es hecho usando la sintaxis @<nombre-de-atributo>.setter (`@nombre.setter`) para el método *setter* y @<nombre-de-atributo>.deleter (`@nombre.deleter`) para el método *deleter*.

Note, es usando el mismo método `nombre()` con diferentes definiciones para definir los métodos `getter`, `setter` y `deleter`.

Ahora, cada vez que se usa `persona1.nombre`, es internamente llama el apropiado método para `getter`, `setter` y `deleter` como lo muestra la salida impresa presente dentro de cada método.

super

La clase `super` típicamente es usada al llamar un método de superclase cooperativo. Las sintaxis de como usarlo son las siguientes:

```
>>> super(type, obj)
```

El código anterior devuelve un súper objeto enlazado; requiere `isinstance(obj, type)`.

```
>>> super(type)
```

El código anterior devuelve un súper objeto no unido.

```
>>> super(type, type2)
```

El código anterior devuelve un súper objeto enlazado; requiere `issubclass(type2, type)`.

Para declarar un método de superclase cooperativo, use esta sintaxis:

```
class ClaseBase():
    def metodo(self, argumento):
        pass
class Clase(ClaseBase):
    def metodo(self, argumento):
        super(Clase, self).metodo(argumento)
```

Un ejemplo sencillo real se muestra a continuación:

```
>>> class Mamifero(object):
...     def __init__(self, mamifero):
...         print mamifero, 'es un animal de sangre caliente.'
...
>>> class Perro(Mamifero):
...     def __init__(self):
...         print 'Perro tiene 4 piernas.'
...         super(Perro, self).__init__('Perro')
...
>>> perrito = Perro()
Perro tiene 4 piernas.
Perro es un animal de sangre caliente.
>>> isinstance(perrito, Perro)
True
```

Aquí, se llama el método `__init__` de la clase `Mamifero` (desde la clase `Perro`) usando el código fuente `super(Perro, self).__init__('Perro')` en vez de del tradicional `Mamifero.__init__(self, 'Perro')`.

Como no necesitamos especificar el nombre de la clase base si usamos `super()`, podemos cambiar fácilmente la clase base para el método `Perro` (si es necesario).

A continuación un ejemplo de cambiar la clase base a la clase RazaCanina:

```
>>> class Mamifero(object):
...     def __init__(self, mamifero):
...         print mamifero, 'es un animal de sangre caliente.'
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> class RazaCanina(Mamifero):
...   def __init__(self, nombre, raza):
...     print raza, 'es la raza del canino.'
...     super(RazaCanina, self).__init__('Perro')
...
>>> class Perro(RazaCanina):
...     def __init__(self, raza):
...         print 'Perro tiene 4 piernas.'
...         super(Perro, self).__init__('Perro', raza)
...
>>> perrito = Perro("Pastor Alemán")
Perro tiene 4 piernas.
Pastor Alemán es la raza del canino.
Perro es un animal de sangre caliente.
```

El método integrado `super()` regresa un objeto proxy, un objeto substituto que tiene la habilidad de llamar al método de la clase base vía delegación. Esto es llamado indirección (habilidad de referenciar objeto base con el método `super()`).

Desde que la indirección es calculada en tiempo ejecución, usted puede usar para apuntar hacia una clase base diferente en tiempo diferente (si usted lo necesita).

A continuación un ejemplo del uso `super()` con *herencia múltiple* (página 205) de la objetos:

```
>>> class Animal(object):
...     def __init__(self, animal):
...         print animal, 'es un animal.\n\n',
...
>>> class Mamifero(Animal):
...     def __init__(self, mamifero):
...         print mamifero, 'es un animal de sangre caliente.'
...         super(Mamifero, self).__init__(mamifero)
...
>>> class MamiferoNoVolador(Mamifero):
...     def __init__(self, mamifero):
...         print mamifero, "no puede volar."
...         super(MamiferoNoVolador, self).__init__(mamifero)
...
>>> class MamiferoNoAcuatico(Mamifero):
...     def __init__(self, mamifero):
...         print mamifero, "no puede nadar."
...         super(MamiferoNoAcuatico, self).__init__(mamifero)
...
>>> class Perro(MamiferoNoAcuatico, MamiferoNoVolador):
...     def __init__(self):
...         print 'Perro tiene 4 piernas.\n',
...         super(Perro, self).__init__('Perro')
...
>>> perro = Perro()
Perro tiene 4 piernas.
Perro no puede nadar.
Perro no puede volar.
Perro es un animal de sangre caliente.
Perro es un animal.

>>> Perro.__mro__
(<class '__main__.Perro'>,
<class '__main__.MamiferoNoAcuatico'>,
<class '__main__.MamiferoNoVolador'>,
<class '__main__.Mamifero'>,
<class '__main__.Animal'>,
<type 'object'>)
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> murcielago = MamiferoNoAcuatico('Murcielago')
Murcielago no puede nadar.
Murcielago es un animal de sangre caliente.
Murcielago es un animal.

>>> MamiferoNoAcuatico.__mro__
(<class '__main__.MamiferoNoAcuatico'>,
<class '__main__.Mamifero'>,
<class '__main__.Animal'>,
<type 'object'>)
```

El orden en resolver la herencia múltiple esta basado en el principio *Method Resolution Order (MRO)* (página 206).

El *MRO* es calculado en Python de la siguiente forma:

Un método en la llamada derivada es siempre llamada antes de método de la clase base. En nuestro ejemplo, la clase `Perro` es llamada antes de las clases `MamiferoNoAcuatico` o `MamiferoNoVolador`. Esas dos clases son llamada antes de la clase `Mamifero` el cual es llamada antes de la clase `Animal` y la clase `Animal` es llamada antes de la clase `object`.

Si hay herencia múltiple como `Perro(MamiferoNoAcuatico, MamiferoNoVolador)`, el método de `MamiferoNoAcuatico` es invocado primero por que ese aparece primero.

9.8.5 type

Los *objetos tipo* (página 26) representan los diversos tipos de objeto. El tipo de un objeto es accesible mediante la función integrada *type()* (página 122). No hay operaciones especiales sobre los tipos. El módulo estándar `types` define nombres para todos los tipos internos estándar.

```
>>> type(type)
<type 'type'>
```

Ver también:

Consulte la sección de *lecturas suplementarias* (página 265) del entrenamiento para ampliar su conocimiento en esta temática.

CAPÍTULO 10

Decoradores y la librería estándar

10.1 Decoradores

Un decorador es una función Python permite que agregar funcionalidad a otra función, pero sin modificarla. También, esto es llamado meta-programación, por que parte del programa trata de modificar a otro al momento de compilar.

Para llamar un decorador se utiliza el signo de arroba (@).

Los decoradores en Python son discutidos y definidos en el PEP-318. https://www.python.org/dev/peps/pep-0318/

Por hacer: TODO Terminar de escribir esta sección

10.2 Listas de comprensión

La listas de comprensión, del inglés *list comprehensions*, es una funcionalidad que le permite crear listas avanzadas en una misma línea de código.

La forma general de la definición de una lista por comprensión es:

```
[expresion for item in iterable]
```

Opcionalmente, se puede incluir un condicional en la expresión:

```
[expresion for item in iterable if condicion]
```

`expresion` puede ser cualquier expresión computable en Python, generalmente involucrando un `item` del iterable llamado `iterable` puede ser cualquier objeto iterable, como una secuencia (*lista* (página 58) o *cadena de caracteres* (página 46)), la función la función *range()* (página 120), etc.

La salida siempre es un tipo de *lista* (página 58) Python.

10.2.1 Ejemplo 1

Si requiere crear una lista de 4 elementos y cada elemento calcularle la potencia de 2, usando el método tradicional, eso seria así:

```
>>> lista = []
>>> for i in range(4):
...     lista.append(i**2)
...
>>> print lista
[0, 1, 4, 9]
```

Entonces el ejemplo anterior usando listas de comprensión, eso seria así:

```
>>> [i**2 for i in range(4)]
[0, 1, 4, 9]
```

10.2.2 Ejemplo 2

A continuación se crear una lista con las letras de una palabra, usando el método tradicional, eso seria así:

```
>>> lista = []
>>> for letra in 'casa':
...     lista.append(letra)
...
>>> print lista
['c', 'a', 's', 'a']
```

Entonces el ejemplo anterior usando listas de comprensión, eso seria así:

```
>>> lista = [letra for letra in 'casa']
>>> print lista
['c', 'a', 's', 'a']
```

Como puede detallar en el ejemplo anterior, gracias a la listas de comprensión usted puede indicar directamente cada elemento que va a formar la lista, en este caso cada letra, a la vez que definimos el *bucle for* (página 90), entonces la lista está formada por cada letra que recorremos en el bucle `for`.

10.2.3 Ejemplo 3

A continuación se crear una lista con las potencias de 2 de los primeros 10 números, usando el método tradicional, eso seria así:

```
>>> lista = []
>>> for numero in range(0,11):
...     lista.append(numero**2)
...
>>> print lista
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Entonces el ejemplo anterior usando listas de comprensión, eso seria así:

```
>>> lista = [numero**2 for numero in range(0,11)]
>>> print lista
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

De este código anterior usted puede aprender que es posible modificar al vuelo los elementos los cuales van a formar la lista.

10.2.4 Ejemplo 4

A continuación se crear una lista con los todos los múltiples de 2 entre 0 y 10, usando el método tradicional, eso seria así:

```
>>> lista = []
>>> for numero in range(0,11):
...     lista.append(numero**2)
...
>>> print lista
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

También, si añade al código anterior, los números del 0 al 10 cuando su módulo de 2 sea 0 usando el método tradicional, eso seria así:

```
>>> lista = []
>>> for numero in range(0,11):
...     if numero % 2 == 0:
...         lista.append(numero)
...
>>> print lista
[0, 2, 4, 6, 8, 10]
```

Entonces el ejemplo anterior donde crear una lista con los todos los múltiples de 2 entre 0 y 10, usando listas de comprensión, eso seria así:

```
>>> lista = [numero for numero in range(0,11) if numero % 2 == 0 ]
>>> print lista
[0, 2, 4, 6, 8, 10]
```

Para el ejemplo anterior donde crear una lista con los todos los múltiples de 2 entre 0 y 10 cuando su módulo de 2 sea 0, usando listas de comprensión, eso seria así:

```
>>> [numero for numero in range(0,11) if numero % 2 == 0 ]
[0, 2, 4, 6, 8, 10]
```

En este caso puede observar que incluso puede marcar una condición justo al final para añadir o no el elemento en la lista.

10.2.5 Ejemplo 5

A continuación se crear una lista de pares a partir de otra lista creada con las potencias de 2 de los primeros 10 números, usando el método tradicional, eso seria así:

```
>>> lista = []
>>> for numero in range(0,11):
...     lista.append(numero**2)
...
>>> pares = []
>>> for numero in lista:
...     if numero % 2 == 0:
...         pares.append(numero)
...
>>> print pares
[0, 4, 16, 36, 64, 100]
```

Entonces el ejemplo anterior usando listas de comprensión, eso seria así:

```
>>> lista = [numero for numero in
...              [numero**2 for numero in range(0,11)]
```

(continué en la próxima página)

(proviene de la página anterior)

```
...                 if numero % 2 == 0]
>>> print lista
[0, 4, 16, 36, 64, 100]
```

Crear listas a partir de listas anidadas le permite llevar la listas de comprensión al siguiente nivel y además no hay un límite.

10.2.6 Usando Listas de comprensión con Archivos

Por hacer: TODO escribir esta sección.

Ver también:

Consulte la sección de *lecturas suplementarias* (página 266) del entrenamiento para ampliar su conocimiento en esta temática.

10.3 La librería estándar Python

La librería estándar Python 2 incluye los siguientes módulos y librerías:

- *Funciones integradas* (página 113) y funciones integradas no esenciales
- Constantes integradas, con el módulo `site`.
- Tipos integrados, incluye 13 tipos, como `Booleano`, `Numérico` y otros.
- Manejo de Excepciones
- Servicios cadenas de caracteres, incluye 11 librerías, como `string`, `re` y otros.
- Tipos de datos, incluye 19 librerías, como `datetime`, `pprint` y otros.
- Módulos numéricos y matemáticos, incluye 9 librerías, como `decimal`, `math` y otros.
- Acceso a archivos y directorios, incluye 12 librerías, como `os.path`, `fileinput` y otros.
- Persistencia de datos, incluye 13 librerías, como `pickle`, `sqlite3` y otros.
- Compresión de datos y de archivo, incluye 5 librerías, como `zlib`, `gzip` y otros.
- Formatos de archivo, incluye 6 librerías, como `csv`, `ConfigParser` y otros.
- Servicios criptográficos, incluye 4 librerías, como `hashlib`, `md5` y otros.
- Servicios genéricos del sistema operativo, incluye 17 librerías, como `os`, `time` y otros.
- Servicios opcionales del sistema operativo, incluye 9 librerías, como `threading`, `readline` y otros.
- Comunicación entre procesos y redes, incluye 7 librerías, como `subprocess`, `socket` y otros.
- Manejo de datos de Internet, incluye 16 librerías, como `email`, `json` y otros.
- Procesamiento de marcado estructurado, incluye 15 librerías, como `HTMLParser`, `htmllib` y otros.
- Protocolos de Internet y soporte, incluye 25 librerías, como `cgi`, `wsgiref` y otros.
- Servicios multimedia, incluye 10 librerías, como `audioop`, `wave` y otros.
- Internacionalización, incluye las librerías `gettext` y `locale`.
- Program Frameworks, incluye las librerías `cmd` y `shlex`.
- Interfaces gráficas de usuario con Tk, incluye 7 librerías, como `Tkinter`, `IDLE` y otros.

- Herramientas de desarrollo, incluye 6 librerías, como `unittest`, `test` y otros.
- Depuración y Profiling, incluye 7 librerías, como `pdb`, `trace` y otros.
- Empaquetado y distribución de software, incluye las librerías `distutils` y `ensurepip`.
- Python Runtime Services, incluye 16 librerías, como `sys`, `site` y otros.
- Intérpretes de Python personalizados, incluye las librerías `code` y `codeop`.
- Ejecución restringida, incluye las librerías `rexec` y `Bastion`.
- Importación de módulos, incluye 7 librerías, como `imp`, `runpy` y otros.
- Python Language Services, incluye 13 librerías, como `parser`, `dis` y otros.
- Paquete compilador de Python, incluye 5 librerías.
- Servicios Misceláneos, `formatter` librería incluida.
- Servicios específicos de MS Windows, incluye 4 librerías, como `msilib`, `winsound` y otros.
- Servicios específicos de Unix, incluye 16 librerías, como `commands`, `syslog` y otros.
- Servicios específicos de Mac OS X, incluye 9 librerías, como `ic`, `MacOS` y otros.
- Módulos de MacPython OSA, incluye 12 librerías, como `aepack`, `aetypes` y otros.
- Servicios específicos de SGI IRIX, incluye 12 librerías, como `gl`, `jpeg` y otros.
- Servicios específicos de SunOS, las librerías `sunaudiodev` y `SUNAUDIODEV`.

Por hacer: TODO terminar de escribir esta sección.

10.4 datetime

Fecha y hora

Por hacer: TODO escribir esta sección.

Advertencia: Tenga en cuenta que este documento no está completo sin la palabra hablada de un instructor a pesar de que tratamos de incluir las partes más importantes de lo que enseñamos en la narrativa no puede considerarse completa sin la palabra hablada.

CAPÍTULO 11

Apéndices

11.1 Esquema del entrenamiento

Este entrenamiento toma 10 lecciones. Cada lección contiene material de lectura y ejercicios que usted tendrá que escribir en el interprete Python. Cada lección aprendida están asociadas entre si mismas.

11.1.1 Lección 1 - Introducción al lenguaje Python

Descripción: Sensibilizar sobre la filosofía del lenguaje, su historia y evolución, casos de éxitos.

Práctica: Exponer los fundamentos sobre el lenguaje Python, comentar sobre usos e implementaciones exitosas a nivel regional, nivel nacional y nivel mundial.

11.1.2 Lección 2 - Introspección del lenguaje Python

Descripción: Conocer las capacidades de introspección que ofrece el lenguaje.

Práctica: Acceder al interprete Python demostrando la documentación propia integrada, analizar las estructuras de datos, métodos, clases y demás elementos disponibles del lenguaje. Instalar el paquete `ipython` y conocer sus ventajas.

11.1.3 Lección 3 - Tipos y estructuras de datos

Descripción: Comprender la creación y asignación de tipos primitivos (variables numéricas, cadenas de texto con sus operaciones; tipos compuestos (listas, tuplas, diccionarios).

Práctica: Ejemplos de creación y asignación de variables numéricas, cadenas de texto, listas, tuplas, diccionarios y explorar el resultado desde el interprete Python.

11.1.4 Lección 4 - Bloques de código y estructuras de control

Descripción: Comprender las estructuras de control como `if` (`elif`, `else`); `for`, `while` (`else`, `break`, `continue`, `pass`); las funciones *range()* (página 120) y *xrange()* (página 121); ademas de los tipos *iteradores()* (página 92).

Práctica: Ejemplos de creación a estructuras condicionales, repetitivas y funciones propias y explorar el resultado desde el interprete Python.

11.1.5 Lección 5 - Funciones y programación estructurada

Descripción: Comprender el uso de las funciones y el paradigma de programación estructurada.

Práctica: Ejemplos de creación e uso de funciones, programar estructuradamente y explorar el resultado desde el interprete Python.

11.1.6 Lección 6 - Introspección a la depuración con pdb

Descripción: Conocer las capacidades de depuración que ofrece el lenguaje.

Práctica: Ejemplo de uso de la herramienta `pdb` y explorar el resultado en un módulo.

11.1.7 Lección 7 - Operaciones de E/S y manipulación de archivos

Descripción: Comprender las operaciones de entrada/salida y manipular archivos.

Práctica: Ejemplo de la sentencia `input`, `raw_input` y `print`, además la creación, apertura, lectura, escritura archivos y explorar el resultado en un módulo.

11.1.8 Lección 8 - Módulos, paquetes y distribución de software

Descripción: Comprender la creación de módulos, paquetes y distribución de software Python e implementación de estos en sus propios desarrollos.

Práctica: Ejemplo de creación de módulos, paquetes y distribución de software Python organizando su código en estas estructuras de programas y explorar el resultado en un módulo y paquete en el sistema de archivos.

11.1.9 Lección 9 - Manejos de errores y orientación a objetos

Descripción: Comprender el manejos de errores (`try`, `except`, `else`, `finally`, y `raise`) y el paradigma de programación orientada a objetos (clases, herencia simple y múltiple, sobrecarga de métodos).

Práctica: Ejemplo de creación de clases, atributos, comportamientos, manipulación de errores en Python y explorar el resultado en un paquete en el sistema de archivos.

11.1.10 Lección 10 - Decoradores y la librería estándar

Descripción: Comprender el uso de decoradores y los diversos módulos adicionales de la librería estándar de Python.

Práctica: Ejemplo de uso de decoradores y módulos adicionales útiles de la biblioteca estándar explorando el resultado en un paquete Python en el sistema de archivos.

11.2 Lecturas suplementarias del entrenamiento

Siempre aprender un nuevo lenguaje de programación tiene nuevos retos desde aprender sobre la filosofía del lenguaje y hasta léxicos propios los cuales hacen característico el uso y expresión de sus programas con este nuevo lenguaje, más esto requiere práctica y tiempo para lograr la fluidez en hablar y escribir programas en Python.

Hay más contenido complementario o detallada que el contenido del entrenamiento de las 10 lecciones cubiertas en este entrenamiento. Al finalizar, yo he compilado una lista de lecturas relacionadas que soporta la información

que usted aprende en cada lección. Piensa que esto como materiales suplementarios. Usted puede leerlo en su tiempo libre para ayudar a incrementar tanto la profundidad y amplitud en su conocimiento.

Las lecturas están organizada como las lecciones y sus temas.

11.2.1 Lección 1 - Introducción al lenguaje Python

Introducción a Python

- Pagina Web Oficial[99].
- Documentación oficial de Python 2.7[100].
- Tutorial de Python 2.7[101].
- Python para programadores con experiencia[102].
- Introducción a la programación con Python[103].
- Python Tutorial[104].
- Wikipedia - Python[105].
- Ver la Figura 4.1 anexo sobre la *introducción al Lenguaje de Programación*.

Instalación

- Descarga Python[106].
- PyPI - the Python Package Index[107].
- Ver la Figura 4.2 anexo sobre la *instalación*.

Su primer programa

- Getting Started with Python[108].
- Ver la Figura 4.3 anexo sobre *su primer programa*.

11.2.2 Lección 2 - Introspección del lenguaje Python

Inmersión al modo interactivo de Python

- Una pequeña inmersión al modo interactivo de Python[109] de la fundación Cenditel.
- A Guide To Object Introspection in Python[110].
- Inmersión en Python[111].
- La librería estándar de Python[112].

[99] https://www.python.org/
[100] https://docs.python.org/2.7/
[101] http://docs.python.org.ar/tutorial/2/contenido.html
[102] http://es.diveintopython.net/odbchelper_divein.html
[103] http://www.mclibre.org/consultar/python/
[104] http://www.tutorialspoint.com/python/index.htm
[105] https://es.wikipedia.org/wiki/Python
[106] https://www.python.org/downloads/
[107] https://pypi.org/
[108] http://www.cs.utexas.edu/~mitra/bytes/start.html
[109] https://lcaballero.wordpress.com/2012/07/01/inmersion-al-modo-interactivo-de-python/
[110] https://www.zeolearn.com/magazine/a-guide-to-object-introspection-in-python
[111] https://diveintopython3.net/
[112] https://docs.python.org/2/library/index.html

- Guía de aprendizaje de Python[113].

11.2.3 Lección 3 - Tipos y estructuras de datos

- Python - Tipos básicos[114].
- Python Data Types[115].

Variables y constantes

- Python Variables, Constants and Literals[116].
- Built-in Constants — Python 3.7 documentation[117].

Operadores aritméticos, tipo enteros y reales

- orden de precedencia - operadores aritméticos[118].
- Introducción a la programación en Python - clase 1[119].
- Ver la Figura 4.4 anexo sobre los *operadores aritméticos, tipo enteros y reales*.

Tipo booleanos y cadenas de caracteres

- Ver la Figura 4.5 anexo sobre los *tipo booleanos y cadenas de caracteres*.

Tipo listas

- Iterando sobre una secuencia - Scipy lecture notes[120].
- Listas I - clase 3 - Introducción a la programación en Python[121].

Tipo tuplas

- Seguimiento de una enumeración - Scipy lecture notes[122].

Tipo diccionarios

- Bucle sobre un diccionario - Scipy lecture notes[123].

11.2.4 Lección 4 - Bloques de código y estructuras de control

Condicional if

- Python - Tipos básicos[124].

[113] http://pyspanishdoc.sourceforge.net/tut/tut.html
[114] http://mundogeek.net/archivos/2008/01/17/python-tipos-basicos/
[115] https://www.programiz.com/python-programming/variables-datatypes
[116] https://www.programiz.com/python-programming/variables-constants-literals
[117] https://docs.python.org/es/3.7/library/constants.html
[118] https://www.eumus.edu.uy/eme/ensenanza/electivas/python/2014/CursoPython_clase01.html#orden-de-precedencia
[119] https://www.eumus.edu.uy/eme/ensenanza/electivas/python/2014/CursoPython_clase01.html
[120] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#iterando-sobre-una-secuencia
[121] https://www.eumus.edu.uy/eme/ensenanza/electivas/python/2014/CursoPython_clase03.html#Listas-I
[122] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#seguimiento-de-una-enumeracion
[123] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#bucle-sobre-un-diccionario
[124] http://mundogeek.net/archivos/2008/01/17/python-tipos-basicos/

- Operadores básicos de Python[125].
- Sentencias IF[126].
- Condicionales if y else en Python[127].
- Expresiones condicionales - Scipy lecture notes[128].
- Ver la Figura 4.6 anexo sobre las *sentencias condicionales*.

Operadores lógicos

- Ver la Figura 4.5 anexo sobre los *tipo booleanos y cadenas de caracteres*.

Bucle while

- Introducción a Bucles "while"[129].
- Ciclo while en Python[130].
- Ver la Figura 4.7 anexo sobre los *bucles*.

Bucle for

- Introducción a Bucles "for"[131].
- Ver la Figura 4.7 anexo sobre los *bucles*.

11.2.5 Lección 5 - Funciones y programación estructurada

Funciones definidas por el usuario

- Introducción a Funciones[132] - ¿Por qué?.
- Definiendo una función - Scipy lecture notes[133].
- Funciones de orden superior[134].
- Ver la Figura 4.8 anexo sobre las *funciones*.

Programación estructurada

- Reusando código: scripts y módulos - Scipy lecture notes[135].
- Programación estructurada[136].
- Paseo por la programación estructurada y modular con Python - Rosalía Peña Ros[137].

[125] http://codigoprogramacion.com/cursos/tutoriales-python/operadores-basicos-de-python.html
[126] http://docs.python.org.ar/tutorial/2/controlflow.html#la-sentencia-if
[127] http://codigoprogramacion.com/cursos/tutoriales-python/condicionales-if-y-else-en-python.html
[128] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#expresiones-condicionales
[129] http://docs.python.org.ar/tutorial/2/introduction.html#primeros-pasos-hacia-la-programacion
[130] http://codigoprogramacion.com/cursos/tutoriales-python/ciclo-while-en-python.html
[131] http://docs.python.org.ar/tutorial/2/controlflow.html#la-sentencia-for
[132] http://docs.python.org.ar/tutorial/2/controlflow.html#definiendo-funciones
[133] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/functions.html#definiendo-una-funcion
[134] https://github.com/josuemontano/python_intro/wiki/Funciones-de-orden-superior
[135] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/reusing_code.html
[136] https://es.wikipedia.org/wiki/Programación_estructurada
[137] https://www.scribd.com/document/545079783/articulo-paseo

11.2.6 Lección 6 - Introspección a la depuración con pdb

- pdb — The Python Debugger[138].
- Usando el depurador Python - Python Scientific Lecture Notes (Spanish translation)[139].
- Ver la Figura 4.9 anexo sobre el *Python Debugger*.

11.2.7 Lección 7 - Operaciones de E/S y manipulación de archivos

Entrada / Salida en Python

- Python Programming / Input and Output[140].
- Python - Entrada / Salida. Ficheros[141].
- Ver la Figura 4.10 anexo sobre la *entrada Estándar rawInput*.
- Ver la Figura 4.11 anexo sobre la *salida Estándar rawInput*.

Manipulación de archivos

- Entrada y Salida - Scipy lecture notes[142].

11.2.8 Lección 8 - Módulos, paquetes y distribución de software

Módulos Python

- Reusando código: scripts y módulos - Scipy lecture notes[143].

Distribución de Software

- Packaging Python Projects[144].

Scaffolding en proyectos Python

- Gestión de proyectos con Buildout, instalando Zope/Plone con este mecanismo[145] desde la comunidad de Plone Venezuela.

11.2.9 Lección 9 - Manejos de errores y orientación a objetos

Errores y excepciones

- Principales errores al comenzar con Python[146].

[138] https://docs.python.org/2/library/pdb.html
[139] https://claudiovz.github.io/scipy-lecture-notes-ES/advanced/debugging/index.html#usando-el-depurador-python
[140] https://en.wikibooks.org/wiki/Python_Programming/Input_and_Output
[141] http://mundogeek.net/archivos/2008/04/02/python-entrada-salida-ficheros/
[142] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/io.html
[143] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/reusing_code.html
[144] https://packaging.python.org/en/latest/tutorials/packaging-projects/
[145] https://coactivate.org/projects/ploneve/gestion-de-proyectos-con-buildout
[146] http://www.cursosgis.com/principales-errores-al-comenzar-con-python/

Programación orientada a objetos

- Programación orientada a objetos - Wikipedia[147].
- Clases — Tutorial de Python v2.7.0[148].
- Programación Orientada a Objetos (POO) - Scipy lecture notes[149].
- What's the meaning of underscores (_ & __) in Python variable names?[150].
- What is the meaning of a single and a double underscore before an object name?[151].
- Ver la Figura 4.12 anexo sobre *Clases y Objetos*.

11.2.10 Lección 10 - Decoradores y la librería estándar

Iteradores

- Ver el uso de *comprensión de listas* (página 266).
- Ver la Figura 4.14 anexo sobre *Generadores*.
- Ver la Figura 4.15 anexo sobre *Decoradores*.

Listas de comprensión

- Listas por comprensión - Scipy lecture notes[152].
- Ver la Figura 4.13 anexo sobre *Comprensión de Listas*.

11.3 Anexos del entrenamiento

A continuación varios material multimedia anexos a las lecciones del entrenamiento:

Figura 11.1: Vídeo Tutorial Python 1 - Introducción al Lenguaje de Programación[153], cortesía de CodigoFacilito.com[154].

[147] https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos
[148] http://docs.python.org.ar/tutorial/2/classes.html
[149] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/oop.html
[150] https://www.youtube.com/watch?v=ALZmCy2u0jQ
[151] https://stackoverflow.com/questions/1301346/what-is-the-meaning-of-a-single-and-a-double-underscore-before-an-object-name
[152] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#listas-por-comprension
[153] https://www.youtube.com/watch?v=CjmzDHMHxwU
[154] https://codigofacilito.com/
[155] https://www.youtube.com/watch?v=VTykmP-a2KY
[156] https://codigofacilito.com/
[157] https://www.youtube.com/watch?v=OtJEj7N9T6k
[158] https://codigofacilito.com/
[159] https://www.youtube.com/watch?v=ssnkfbBbcuw
[160] https://codigofacilito.com/

Figura 11.2: Vídeo Tutorial Python 2 - Instalación[155], cortesía de CodigoFacilito.com[156].

Figura 11.3: Vídeo Tutorial Python 3 - Hola Mundo[157], cortesía de CodigoFacilito.com[158].

Figura 11.4: Vídeo Tutorial Python 4 - Enteros, reales y operadores aritméticos[159], cortesía de CodigoFacilito.com[160].

Figura 11.5: Vídeo Tutorial Python 5 - Booleanos, operadores lógicos y cadenas[161], cortesía de CodigoFacilito.com[162].

Figura 11.6: Vídeo Tutorial Python 10 - Sentencias condicionales[163], cortesía de CodigoFacilito.com[164].

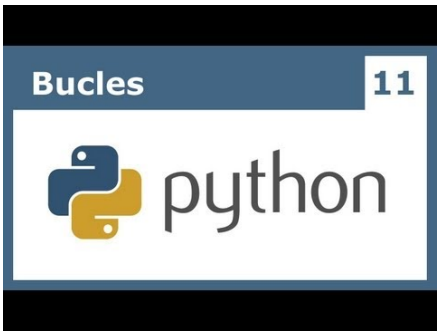

Figura 11.7: Vídeo Tutorial Python 11 - Bucles[165], cortesía de CodigoFacilito.com[166].

Figura 11.8: Vídeo Tutorial Python 12 - Funciones[167], cortesía de CodigoFacilito.com[168].

[161] https://www.youtube.com/watch?v=ZrxcqbFYjiw
[162] https://codigofacilito.com/
[163] https://www.youtube.com/watch?v=hLqKvB7tGWk
[164] https://codigofacilito.com/
[165] https://www.youtube.com/watch?v=IyI2ZuOq_xQ
[166] https://codigofacilito.com/
[167] https://www.youtube.com/watch?v=_C7Uj7O5o_Q
[168] https://codigofacilito.com/
[169] https://www.youtube.com/watch?v=N4NtB4r28h0
[170] https://www.youtube.com/watch?v=AzeUCuMvW6I
[171] https://codigofacilito.com/
[172] https://www.youtube.com/watch?v=B-JPXgxK3Oc
[173] https://codigofacilito.com/
[174] https://www.youtube.com/watch?v=VYXdpjCZojA
[175] https://codigofacilito.com/
[176] https://www.youtube.com/watch?v=87s8XQbUv1k
[177] https://codigofacilito.com/
[178] https://www.youtube.com/watch?v=tvHbC_OZV14
[179] https://codigofacilito.com/
[180] https://www.youtube.com/watch?v=TaIWx9paNIA
[181] https://codigofacilito.com/

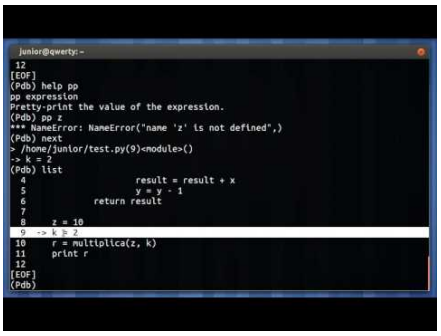

Figura 11.9: Vídeo Depurando um programa Python com pdb - Python Debugger[169], cortesía de *Youtube*.

Figura 11.10: Vídeo Tutorial Python 30 - Entrada Estándar rawInput[170], cortesía de CodigoFacilito.com[171].

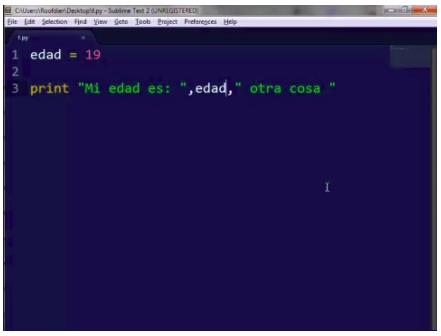

Figura 11.11: Vídeo Tutorial Python 31 - Salida Estándar rawInput[172], cortesía de CodigoFacilito.com[173].

Figura 11.12: Vídeo Tutorial Python 13 - Clases y Objetos[174], cortesía de CodigoFacilito.com[175].

Figura 11.13: Vídeo Tutorial Python 25 - Comprensión de Listas[176], cortesía de CodigoFacilito.com[177].

Figura 11.14: Vídeo Tutorial Python 26 - Generadores[178], cortesía de CodigoFacilito.com[179].

Figura 11.15: Vídeo Tutorial Python 27 - Decoradores[180], cortesía de CodigoFacilito.com[181].

11.4 Operadores

A continuación, se ofrecen una referencia corta de los diversos *operadores* en Python:

11.4.1 Operadores de asignaciones

Una corta referencia de los *operadores de asignación* se ofrece a continuación:

Operador	Descripción	Ejemplo
=	asigna valor a una variable	`>>> r = 5` `>>> r1 = r`

+=	suma el valor a la variable	`>>> r = 5`
		`>>> r += 10; r`
		`15`

-=	resta el valor a la variable	`>>> r = 5`
		`>>> r -= 10; r`
		`-5`

*=	multiplica el valor a la variable	`>>> r = 5`
		`>>> r *= 10; r`
		`50`

/=	divide el valor a la variable	`>>> r = 5`
		`>>> r /= 10; r`
		`0`

**=	calcula el exponente del valor de la variable	`>>> r = 5`
		`>>> r **= 10; r`
		`9765625`

//=	calcula la división entera del valor de la variable	`>>> r = 5`
		`>>> r //= 10; r`
		`0`

%=	devuelve el resto de la división del valor de la variable	`>>> r = 5`
		`>>> r %= 10; r`
		`5`

Ver también:

Usted puede consultar para más información la explicación de cada uno de los *operadores de asignaciones* (página 32).

11.4.2 Operadores aritméticos

Una corta referencia de los *operadores aritméticos* se ofrece a continuación:

Operador	Descripción	Ejemplo
+	Suma	`>>> 3 + 2` `5`
-	Resta	`>>> 4 - 7` `-3`
-	Negación	`>>> -7` `-7`
*	Multiplicación	`>>> 2 * 6` `12`
**	Exponente	`>>> 2 ** 6` `64`
/	División	`>>> 3.5 / 2` `1.75`
//	División entera	`>>> 3.5 // 2` `1.0`
%	Módulo	`>>> 7 % 2` `1`

Ver también:

Usted puede consultar para más información la explicación de cada uno de los *operadores aritméticos* (página 35).

11.4.3 Operadores relacionales

Una corta referencia de los *operadores relacionales* se ofrece a continuación:

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	`>>> 5 == 3` `False`
!=	¿son distintos a y b?	`>>> 5 != 3` `True`
<	¿es a menor que b?	`>>> 5 < 3` `False`
>	¿es a mayor que b?	`>>> 5 > 3` `True`
<=	¿es a menor o igual que b?	`>>> 5 <= 5` `True`
>=	¿es a mayor o igual que b?	`>>> 5 >= 3` `True`

Ver también:

Usted puede consultar para más información la explicación de cada uno de los *operadores relacionales* (página 38).

11.4.4 Operadores lógicos

Una corta referencia de los *operadores lógicos* se ofrece a continuación:

Operador	Descripción	Ejemplo
and	¿se cumple a y b?	`>>> True and False` `False`
or	¿se cumple a o b?	`>>> True or False` `True`
not	No al valor	`>>> not True` `False`

Ver también:

Usted puede consultar para más información la explicación de cada uno de los *operadores lógicos* (página 86).

11.5 Glosario

Autor(es) Leonardo J. Caballero G.

Correo(s) leonardoc@plone.org

Compatible con Python 2.x, Python 3.x

Fecha 11 de Enero de 2021

A continuación una serie de términos usados en las tecnologías Python/Zope/Plone.

buildout En la herramienta buildout[182], es un conjunto de partes que describe como ensamblar una aplicación.

bundle Ver *Paquete bundle*.

Catalog Sinónimo en Ingles del termino *Catálogo*.

Catálogo Es un índice interno de los contenidos dentro de Plone para que se pueda buscar. El objetivo del catálogo es que sea accesible a través de la ZMI[183] a través de la herramienta portal_catalog[184].

Cheese shop Ver *PyPI*.

Collective Es un repositorio de código comunitario, para Productos Plone y productos de terceros, y es un sitio muy útil para buscar la ultima versión de código fuente del producto para cientos de productos de terceros a Plone. Los desarrolladores de nuevos productos de Plone son animados a compartir su código a través de Collective para que otros puedan encontrarlo, usarlo, y contribuir con correcciones / mejoras.

En la actualidad la comunidad ofrece dos repositorio Collective un basado en **Git** y otro **Subversion**.

Si usted quiere publicar un nuevo producto en el repositorio *Git de Collective* de Plone necesita obtener acceso de escritura[185] y seguir las reglas en github/collective, también puede consultarlo en la cuenta en github.com[186].

Si usted quiere publicar un nuevo producto en el repositorio *Subversion de Collective* de Plone necesita obtener acceso de escritura al repositorio[187] y crear su estructura básica de repositorio[188] para su producto, también puede consultarlo vía Web consulte el siguiente enlace[189].

Declaración ZCML El uso concreto de una *Directiva ZCML* dentro de un archivo *ZCML*.

Directiva ZCML Una «etiqueta» *ZCML* como `<include />` o `<utility />`.

Egg Ver *paquetes Egg*.

esqueleto Los archivos y carpetas recreados por un usuario el cual los genero ejecutando alguna plantilla `templer` (`PasteScript`).

estructura 1) Una clase Python la cual controla la generación de un árbol de carpetas que contiene archivos.

2) Una unidad de carpetas y archivos proveídos por el sistema `templer` para ser usado en una plantilla o plantillas. Las estructuras proporcionan recursos estáticos compartidos, que pueden ser utilizados por cualquier paquete en el sistema de `templer`.

Las estructuras diferencian de las plantillas en que no proporcionan las *vars*.

filesystem Termino ingles File system, referido al sistema de archivo del sistema operativo.

Flujo de trabajo Es una forma muy poderosa de imitar los procesos de negocio de su organización, es también la forma en se manejan la configuración de seguridad de Plone.

Flujo de trabajos Plural del termino *Flujo de trabajo*.

grok Ver la documentacion del proyecto grok[190].

Instalación de Zope El software propio del servidor de aplicaciones.

Instancia de Zope Un directorio específico que contiene una configuración completa del servidor Zope.

[182] https://plone-spanish-docs.readthedocs.io/es/latest/buildout/replicacion_proyectos_python.html
[183] https://plone-spanish-docs.readthedocs.io/es/latest/zope/zmi/index.html
[184] https://plone-spanish-docs.readthedocs.io/es/latest/zope/zmi/index.html#portal-catalog
[185] https://collective.github.io/
[186] https://github.com/collective
[187] https://old.plone.org/countries/conosur/documentacion/obtener-acceso-de-escritura-al-repositorio-svn-de-plone
[188] https://old.plone.org/countries/conosur/documentacion/crear-un-nuevo-proyecto-en-el-repositorio-collective-de-plone
[189] https://svn.plone.org/svn/collective/
[190] https://grok-community-docs.readthedocs.io/en/latest/

local command Una clase Paste[191] la cual provee funcionalidad adicional a una estructura de esqueleto de proyecto que ha sido generada.

módulo Del Ingles `module`, es un archivo fuente Python; un archivo en el sistema de archivo que típicamente finaliza con la extensión `.py` o `.pyc`. Los modules son parte de un *paquete*.

Nombre de puntos Python Es la representación Python del «camino» para un determinado objeto / módulo / función, por ejemplo, `Products.GenericSetup.tool.exportToolset`. A menudo se utiliza como referencia en configuraciones `Paste` y `setuptools` a cosas en Python.

paquete Ver *Paquete Python*.

Paquete bundle Este paquete consististe en un archivo comprimido con todos los módulos que son necesario compilar o instalar en el *PYTHONPATH* de tu interprete `Python`.

paquete Egg Es una forma de empaquetar y distribuir paquetes Python. Cada Egg contiene un archivo `setup.py` con metadata (como el nombre del autor y la correo electrónico y información sobre el licenciamiento), como las dependencias del paquete.

La herramienta del *setuptools <que_es_setuptools>*, es la librería Python que permite usar el mecanismo de paquetes egg, esta es capaz de encontrar y descargar automáticamente las dependencias de los paquetes Egg que se instale.

Incluso es posible que dos paquetes Egg diferentes necesiten utilizar simultáneamente diferentes versiones de la misma dependencia. El formato de paquetes Eggs también soportan una función llamada `entry points`, una especie de mecanismo genérico de plug-in. Mucha más detalle sobre este tema se encuentra disponible en el sitio web de PEAK[192].

Paquete Python Es un termino generalmente usando para describir un módulo Python. en el más básico nivel, un paquete es un directorio que contiene un archivo `__init__.py` y algún código Python.

paquetes Egg Plural del termino *paquete Egg*.

Paquetes Python Plural del termino *Paquete Python*.

part En la herramienta *buildout*, es un conjunto opciones que le permite a usted construir una pieza de la aplicación.

plantilla 1) Una clase Python la cual controla la generación de un esqueleto. Las plantillas contiene una lista de variables para obtener la respuesta de un usuario. Las plantillas son ejecutadas con el comando `templer` suministrando el nombre de la plantilla como un argumento `templer basic_namespace my.package`.

2) Los archivos y carpetas proveídas un paquete `templer` como contenido a ser generado. Las respuestas proporcionadas por un usuario en respuesta a las variables se utilizan para rellenar los marcadores de posición en este contenido.

Producto Es una terminología usada por la comunidad Zope / Plone asociada a cualquier implementación de módulos / complementos y agregados que amplíen la funcionalidad por defecto que ofrece Zope / Plone. También son conocidos como *«Productos de terceros»* del Ingles Third-Party Products[193].

Producto Plone Es un tipo especial de paquete Zope usado para extender las funcionalidades de Plone. Se puede decir que son productos que su ámbito de uso es solo en el desde la interfaz gráfica de Plone.

Producto Zope Es un tipo especial de paquete Python usado para extender Zope. En las antiguas versiones de Zope, todos los productos eran carpetas que se ubican dentro de una carpeta especial llamada `Products` de una instancia Zope; estos tendrían un nombre de módulo Python que empiezan por «**Products.**». Por ejemplo, el núcleo de Plone es un producto llamado `CMFPlone`, conocido en Python como Products.CMFPlone[194].

Este tipo de productos esta disponibles desde la interfaz administrativa de Zope (ZMI)[195] de su instalación[196]

[191] https://paste.readthedocs.io/en/latest/
[192] http://peak.telecommunity.com/DevCenter/setuptools
[193] https://docs.plone.org/develop/addons/
[194] https://pypi.org/project/Products.CMFPlone
[195] https://plone-spanish-docs.readthedocs.io/es/latest/zope/zmi/index.html
[196] http://localhost:8080/manage

donde deben acceder con las credenciales del usuario Administrador de Zope. Muchas veces el producto simplemente no hay que instalarlo por que se agregar automáticamente.

Productos Plural del termino *Producto*.

Productos Plone Plural del termino *Producto Plone*.

Productos Zope Plural del termino *Producto Zope*.

profile Una configuración «predeterminada» de un sitio, que se define en el sistema de archivos o en un archivo tar.

PyPI Siglas del termino en Ingles *Python Package Index*, es el servidor central de *paquetes Egg* Python ubicado en la dirección https://pypi.org/.

Python Package Index Ver *PyPI*.

PYTHONPATH Una lista de nombre de directorios, que contiene librerías Python, con la misma sintaxis como la declarativa `PATH` del shell del sistema operativo.

recipe En la herramienta *buildout*, es el software usado para crear partes de una instalación basada en sus opciones. Más información consulte el articulo Recipes Buildout[197].

setup.py El archivo `setup.py` es un módulo de Python, que por lo general indica que el módulo / paquete que está a punto de instalar ha sido empacado y distribuidos con `Distutils`, que es el estándar para la distribución de módulos de Python.

Con esto le permite instalar fácilmente paquetes de Python, a menudo es suficiente para escribir:

```
python setup.py install
```

Entonces el módulo Python se instalará.

Ver también:

- https://docs.python.org/2/install/index.html

Temas / Apariencias Por lo general si un producto de Tema esta bien diseñado y implementado debe aplicarse de una ves al momento de instalarlo. En caso que no se aplique de una puede acceder a la sección Configuración de Temas[198] y cambiar el **Tema predeterminado** por el de su gusto.

Tipos de contenidos Los tipos de contenidos son productos que extienden la funcionalidad de **Agregar elemento** que permite agregar nuevos tipos de registros (Contenidos) a tu sitio. Esto quiere decir que si instala un tipo de contenido exitosamente debería poder acceder a usarlo desde el menú de **Agregar elemento** en el sitio Plone. Opcionalmente algunos productos instalan un panel de control del producto que puede acceder a este en la sección Configuración de Productos Adicionales[199].

var Diminutivo en singular del termino *variable*.

variable 1) Una pregunta que debe ser respondida por el usuario cuando esta generando una estructura de esqueleto de proyecto usando el sistema de plantilla `templer`. En este caso una variable (var) es una descripción de la información requerida, texto de ayuda y reglas de validación para garantizar la entrada de usuario correcta.

2) Una declarativa cuyo valor puede ser variable o constante dentro de un programa Python o en el sistema operativo.

variables Plural del termino *variable*.

vars Diminutivo en plural del termino *variable*.

Workflow Ver *Flujo de trabajo*.

ZCA

Zope Component Architecture La arquitectura de componentes de Zope (alias ZCA)[200], es un sistema que per-

[197] https://plone-spanish-docs.readthedocs.io/es/latest/buildout/recipes.html
[198] http://localhost:8080/Plone/@@skins-controlpanel
[199] http://localhost:8080/Plone/prefs_install_products_form
[200] https://plone-spanish-docs.readthedocs.io/es/latest/programacion/zca/zca-es.html

mite la aplicación y la expedición enchufabilidad complejo basado en objetos que implementan una interfaz.

ZCatalog Ver *Catalog*.

ZCML Siglas del termino en Ingles *Zope Configuration Mark-up Language*.

ZCML-slug Los así llamados «ZCML-slugs», era configuraciones que estaban destinados a enlazar dentro de un directorio una configuración especial en una instalación de Zope, por lo general se ven como `collective.foo-configure.zcml`. Estas configuraciones ya no están más en uso, pueden ser eliminados agregando las configuraciones del paquete z3c.autoinclude[201].

Zope Configuration Mark-up Language Es un dialecto XML utilizado por Zope para las tareas de configuración. ZCML es capaz de realizar diferentes tipos de declaración de configuración. Es utilizado para extender y conectar a los sistemas basados en la *Zope Component Architecture*.

`Zope 3` tiene la política de separar el código actual y moverlo a los archivos de configuración independientes, típicamente un archivo `configure.zcml` en un buildout. Este archivo configura la instancia Zope. El concepto "Configuración" podría ser un poco engañoso aquí y debe ser pensado o tomarse más cableado.

`ZCML`, el lenguaje de configuración basado en `XML` que se utiliza para esto, se adapta a hacer el registro de componentes y declaraciones de seguridad, en su mayor parte. Al habilitar o deshabilitar ciertos componentes en ZCML, puede configurar ciertas políticas de la aplicación general. En `Zope 2`, habilitar y deshabilitar componentes significa eliminar o remover un determinado producto `Zope 2`. Cuando está ahí, se importa y se carga automáticamente. Este no es el caso en `Zope 3` Si no habilita explícitamente, no va a ser encontrado.

El *grok* proyecto ha adoptado un enfoque diferente para el mismo problema, y permite el registro de componentes, etc haciendo declarativa de código Python. Ambos enfoques son posibles en Plone.

11.6 Licenciamientos

11.6.1 Reconocimiento-CompartirIgual 3.0 Venezuela de Creative Commons

Sobre esta licencia

Esta documentación se distribuye bajo los términos de la licencia Reconocimiento-CompartirIgual 3.0 Venezuela de Creative Commons[202].

Usted es libre de:

- Compartir — copiar y redistribuir el material en cualquier medio o formato.
- Adaptar — remezclar, transformar y crear a partir del material.
- Para cualquier propósito, incluso comercialmente.
- El licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Bajo los siguientes términos:

- Reconocimiento - Usted debe dar el crédito apropiado, proporcionar un enlace a la licencia, y de indicar si se han realizado cambios. Usted puede hacerlo de cualquier manera razonable, pero no en una manera que sugiere el licenciante a usted o que apruebe su utilización.
- CompartirIgual - Si usted mezcla, transforma o crea nuevo material a partir de esta obra, usted podrá distribuir su contribución siempre que utilice la misma licencia que la obra original.

Términos detallados de la licencia: https://creativecommons.org/licenses/by-sa/3.0/ve/legalcode

[201] https://pypi.org/project/z3c.autoinclude
[202] https://creativecommons.org/licenses/by-sa/3.0/ve/

11.7 Tareas pendientes

Esta documentación tiene las siguientes tareas pendientes por hacer para mejorar la calidad de la misma:

Por hacer: TODO Terminar de escribir esta sección

(La entrada original (página 231) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion10/decoradores.rst, línea 17.)

Por hacer: TODO escribir esta sección.

(La entrada original (página 235) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion10/fecha_hora.rst, línea 209.)

Por hacer: TODO terminar de escribir esta sección.

(La entrada original (página 235) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion10/libreria_estandar.rst, línea 87.)

Por hacer: TODO escribir esta sección.

(La entrada original (página 234) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion10/listas_comprension.rst, línea 202.)

Por hacer: TODO terminar de escribir la sección Funciones de predicado.

(La entrada original (página 106) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion5/funciones_avanzadas.rst, línea 21.)

Por hacer: TODO escribir la sección Objetos de función.

(La entrada original (página 106) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion5/funciones_avanzadas.rst, línea 28.)

Por hacer: TODO terminar de escribir la sección Funciones recursivas.

(La entrada original (página 109) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion5/funciones_recursivas.rst, línea 76.)

Por hacer: TODO terminar de escribir esta sección

(La entrada original (página 208) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion9/abstraccion.rst, línea 98.)

Por hacer: TODO escribir sobre esta clase integrada.

(La entrada original (página 212) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion9/clases_integradas.rst, línea 159.)

Por hacer: TODO terminar de escribir sobre la clase integrada classmethod.

(La entrada original (página 222) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion9/clases_integradas.rst, línea 940.)

Por hacer: TODO escribir esta sección

(La entrada original (página 208) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion9/polimorfismo.rst, línea 15.)

Por hacer: TODO explicar el concepto Estado de un objeto.

(La entrada original (página 196) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion9/poo.rst, línea 185.)

Por hacer: TODO explicar por que se lanza la excepción TypeError.

(La entrada original (página 199) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion9/poo.rst, línea 342.)

Por hacer: TODO explicar el concepto de Interfaces.

(La entrada original (página 200) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion9/poo.rst, línea 456.)

Por hacer: TODO explicar el concepto de Implementaciones.

(La entrada original (página 200) se encuentra en /home/docs/checkouts/readthedocs.org/user_builds/entrenamiento-python-basico/checkouts/2.7/source/leccion9/poo.rst, línea 472.)

CAPÍTULO 12

Búsqueda

- search

APÉNDICE A

Esquema del entrenamiento

Este entrenamiento toma 10 lecciones. Cada lección contiene material de lectura y ejercicios que usted tendrá que escribir en el interprete Python. Cada lección aprendida están asociadas entre si mismas.

A.1 Lección 1 - Introducción al lenguaje Python

Descripción: Sensibilizar sobre la filosofía del lenguaje, su historia y evolución, casos de éxitos.

Práctica: Exponer los fundamentos sobre el lenguaje Python, comentar sobre usos e implementaciones exitosas a nivel regional, nivel nacional y nivel mundial.

A.2 Lección 2 - Introspección del lenguaje Python

Descripción: Conocer las capacidades de introspección que ofrece el lenguaje.

Práctica: Acceder al interprete Python demostrando la documentación propia integrada, analizar las estructuras de datos, métodos, clases y demás elementos disponibles del lenguaje. Instalar el paquete `ipython` y conocer sus ventajas.

A.3 Lección 3 - Tipos y estructuras de datos

Descripción: Comprender la creación y asignación de tipos primitivos (variables numéricas, cadenas de texto con sus operaciones; tipos compuestos (listas, tuplas, diccionarios).

Práctica: Ejemplos de creación y asignación de variables numéricas, cadenas de texto, listas, tuplas, diccionarios y explorar el resultado desde el interprete Python.

A.4 Lección 4 - Bloques de código y estructuras de control

Descripción: Comprender las estructuras de control como `if` (`elif`, `else`); `for`, `while` (`else`, `break`, `continue`, `pass`); las funciones *range()* (página 120) y *xrange()* (página 121); ademas de los tipos *iteradores()* (página 92).

Práctica: Ejemplos de creación a estructuras condicionales, repetitivas y funciones propias y explorar el resultado desde el interprete Python.

A.5 Lección 5 - Funciones y programación estructurada

Descripción: Comprender el uso de las funciones y el paradigma de programación estructurada.

Práctica: Ejemplos de creación e uso de funciones, programar estructuradamente y explorar el resultado desde el interprete Python.

A.6 Lección 6 - Introspección a la depuración con pdb

Descripción: Conocer las capacidades de depuración que ofrece el lenguaje.

Práctica: Ejemplo de uso de la herramienta `pdb` y explorar el resultado en un módulo.

A.7 Lección 7 - Operaciones de E/S y manipulación de archivos

Descripción: Comprender las operaciones de entrada/salida y manipular archivos.

Práctica: Ejemplo de la sentencia `input`, `raw_input` y `print`, además la creación, apertura, lectura, escritura archivos y explorar el resultado en un módulo.

A.8 Lección 8 - Módulos, paquetes y distribución de software

Descripción: Comprender la creación de módulos, paquetes y distribución de software Python e implementación de estos en sus propios desarrollos.

Práctica: Ejemplo de creación de módulos, paquetes y distribución de software Python organizando su código en estas estructuras de programas y explorar el resultado en un módulo y paquete en el sistema de archivos.

A.9 Lección 9 - Manejos de errores y orientación a objetos

Descripción: Comprender el manejos de errores (`try`, `except`, `else`, `finally`, y `raise`) y el paradigma de programación orientada a objetos (clases, herencia simple y múltiple, sobrecarga de métodos).

Práctica: Ejemplo de creación de clases, atributos, comportamientos, manipulación de errores en Python y explorar el resultado en un paquete en el sistema de archivos.

A.10 Lección 10 - Decoradores y la librería estándar

Descripción: Comprender el uso de decoradores y los diversos módulos adicionales de la librería estándar de Python.

Práctica: Ejemplo de uso de decoradores y módulos adicionales útiles de la biblioteca estándar explorando el resultado en un paquete Python en el sistema de archivos.

APÉNDICE B

Lecturas suplementarias del entrenamiento

Siempre aprender un nuevo lenguaje de programación tiene nuevos retos desde aprender sobre la filosofía del lenguaje y hasta léxicos propios los cuales hacen característico el uso y expresión de sus programas con este nuevo lenguaje, más esto requiere práctica y tiempo para lograr la fluidez en hablar y escribir programas en Python.

Hay más contenido complementario o detallada que el contenido del entrenamiento de las 10 lecciones cubiertas en este entrenamiento. Al finalizar, yo he compilado una lista de lecturas relacionadas que soporta la información que usted aprende en cada lección. Piensa que esto como materiales suplementarios. Usted puede leerlo en su tiempo libre para ayudar a incrementar tanto la profundidad y amplitud en su conocimiento.

Las lecturas están organizada como las lecciones y sus temas.

B.1 Lección 1 - Introducción al lenguaje Python

B.1.1 Introducción a Python

- Pagina Web Oficial[203].
- Documentación oficial de Python 2.7[204].
- Tutorial de Python 2.7[205].
- Python para programadores con experiencia[206].
- Introducción a la programación con Python[207].
- Python Tutorial[208].
- Wikipedia - Python[209].
- Ver la Figura 4.1 anexo sobre la *introducción al Lenguaje de Programación*.

[203] https://www.python.org/
[204] https://docs.python.org/2.7/
[205] http://docs.python.org.ar/tutorial/2/contenido.html
[206] http://es.diveintopython.net/odbchelper_divein.html
[207] http://www.mclibre.org/consultar/python/
[208] http://www.tutorialspoint.com/python/index.htm
[209] https://es.wikipedia.org/wiki/Python

B.1.2 Instalación

- Descarga Python[210].
- PyPI - the Python Package Index[211].
- Ver la Figura 4.2 anexo sobre la *instalación*.

B.1.3 Su primer programa

- Getting Started with Python[212].
- Ver la Figura 4.3 anexo sobre *su primer programa*.

B.2 Lección 2 - Introspección del lenguaje Python

B.2.1 Inmersión al modo interactivo de Python

- Una pequeña inmersión al modo interactivo de Python[213] de la fundación Cenditel.
- A Guide To Object Introspection in Python[214].
- Inmersión en Python[215].
- La librería estándar de Python[216].
- Guía de aprendizaje de Python[217].

B.3 Lección 3 - Tipos y estructuras de datos

- Python - Tipos básicos[218].
- Python Data Types[219].

B.3.1 Variables y constantes

- Python Variables, Constants and Literals[220].
- Built-in Constants — Python 3.7 documentation[221].

[210] https://www.python.org/downloads/
[211] https://pypi.org/
[212] http://www.cs.utexas.edu/~mitra/bytes/start.html
[213] https://lcaballero.wordpress.com/2012/07/01/inmersion-al-modo-interactivo-de-python/
[214] https://www.zeolearn.com/magazine/a-guide-to-object-introspection-in-python
[215] https://diveintopython3.net/
[216] https://docs.python.org/2/library/index.html
[217] http://pyspanishdoc.sourceforge.net/tut/tut.html
[218] http://mundogeek.net/archivos/2008/01/17/python-tipos-basicos/
[219] https://www.programiz.com/python-programming/variables-datatypes
[220] https://www.programiz.com/python-programming/variables-constants-literals
[221] https://docs.python.org/es/3.7/library/constants.html

B.3.2 Operadores aritméticos, tipo enteros y reales

- orden de precedencia - operadores aritméticos[222].
- Introducción a la programación en Python - clase 1[223].
- Ver la Figura 4.4 anexo sobre los *operadores aritméticos, tipo enteros y reales*.

B.3.3 Tipo booleanos y cadenas de caracteres

- Ver la Figura 4.5 anexo sobre los *tipo booleanos y cadenas de caracteres*.

B.3.4 Tipo listas

- Iterando sobre una secuencia - Scipy lecture notes[224].
- Listas I - clase 3 - Introducción a la programación en Python[225].

B.3.5 Tipo tuplas

- Seguimiento de una enumeración - Scipy lecture notes[226].

B.3.6 Tipo diccionarios

- Bucle sobre un diccionario - Scipy lecture notes[227].

B.4 Lección 4 - Bloques de código y estructuras de control

B.4.1 Condicional if

- Python - Tipos básicos[228].
- Operadores básicos de Python[229].
- Sentencias IF[230].
- Condicionales if y else en Python[231].
- Expresiones condicionales - Scipy lecture notes[232].
- Ver la Figura 4.6 anexo sobre las *sentencias condicionales*.

B.4.2 Operadores lógicos

- Ver la Figura 4.5 anexo sobre los *tipo booleanos y cadenas de caracteres*.

[222] https://www.eumus.edu.uy/eme/ensenanza/electivas/python/2014/CursoPython_clase01.html#orden-de-precedencia
[223] https://www.eumus.edu.uy/eme/ensenanza/electivas/python/2014/CursoPython_clase01.html
[224] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#iterando-sobre-una-secuencia
[225] https://www.eumus.edu.uy/eme/ensenanza/electivas/python/2014/CursoPython_clase03.html#Listas-I
[226] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#seguimiento-de-una-enumeracion
[227] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#bucle-sobre-un-diccionario
[228] http://mundogeek.net/archivos/2008/01/17/python-tipos-basicos/
[229] http://codigoprogramacion.com/cursos/tutoriales-python/operadores-basicos-de-python.html
[230] http://docs.python.org.ar/tutorial/2/controlflow.html#la-sentencia-if
[231] http://codigoprogramacion.com/cursos/tutoriales-python/condicionales-if-y-else-en-python.html
[232] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#expresiones-condicionales

B.4.3 Bucle while

- Introducción a Bucles "while"[233].
- Ciclo while en Python[234].
- Ver la Figura 4.7 anexo sobre los *bucles*.

B.4.4 Bucle for

- Introducción a Bucles "for"[235].
- Ver la Figura 4.7 anexo sobre los *bucles*.

B.5 Lección 5 - Funciones y programación estructurada

B.5.1 Funciones definidas por el usuario

- Introducción a Funciones[236] - ¿Por qué?.
- Definiendo una función - Scipy lecture notes[237].
- Funciones de orden superior[238].
- Ver la Figura 4.8 anexo sobre las *funciones*.

B.5.2 Programación estructurada

- Reusando código: scripts y módulos - Scipy lecture notes[239].
- Programación estructurada[240].
- Paseo por la programación estructurada y modular con Python - Rosalía Peña Ros[241].

B.6 Lección 6 - Introspección a la depuración con pdb

- pdb — The Python Debugger[242].
- Usando el depurador Python - Python Scientific Lecture Notes (Spanish translation)[243].
- Ver la Figura 4.9 anexo sobre el *Python Debugger*.

[233] http://docs.python.org.ar/tutorial/2/introduction.html#primeros-pasos-hacia-la-programacion
[234] http://codigoprogramacion.com/cursos/tutoriales-python/ciclo-while-en-python.html
[235] http://docs.python.org.ar/tutorial/2/controlflow.html#la-sentencia-for
[236] http://docs.python.org.ar/tutorial/2/controlflow.html#definiendo-funciones
[237] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/functions.html#definiendo-una-funcion
[238] https://github.com/josuemontano/python_intro/wiki/Funciones-de-orden-superior
[239] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/reusing_code.html
[240] https://es.wikipedia.org/wiki/Programación_estructurada
[241] https://www.scribd.com/document/545079783/articulo-paseo
[242] https://docs.python.org/2/library/pdb.html
[243] https://claudiovz.github.io/scipy-lecture-notes-ES/advanced/debugging/index.html#usando-el-depurador-python

B.7 Lección 7 - Operaciones de E/S y manipulación de archivos

B.7.1 Entrada / Salida en Python

- Python Programming / Input and Output[244].
- Python - Entrada / Salida. Ficheros[245].
- Ver la Figura 4.10 anexo sobre la *entrada Estándar rawInput*.
- Ver la Figura 4.11 anexo sobre la *salida Estándar rawInput*.

B.7.2 Manipulación de archivos

- Entrada y Salida - Scipy lecture notes[246].

B.8 Lección 8 - Módulos, paquetes y distribución de software

B.8.1 Módulos Python

- Reusando código: scripts y módulos - Scipy lecture notes[247].

B.8.2 Distribución de Software

- Packaging Python Projects[248].

B.8.3 Scaffolding en proyectos Python

- Gestión de proyectos con Buildout, instalando Zope/Plone con este mecanismo[249] desde la comunidad de Plone Venezuela.

B.9 Lección 9 - Manejos de errores y orientación a objetos

B.9.1 Errores y excepciones

- Principales errores al comenzar con Python[250].

B.9.2 Programación orientada a objetos

- Programación orientada a objetos - Wikipedia[251].
- Clases — Tutorial de Python v2.7.0[252].

[244] https://en.wikibooks.org/wiki/Python_Programming/Input_and_Output
[245] http://mundogeek.net/archivos/2008/04/02/python-entrada-salida-ficheros/
[246] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/io.html
[247] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/reusing_code.html
[248] https://packaging.python.org/en/latest/tutorials/packaging-projects/
[249] https://coactivate.org/projects/ploneve/gestion-de-proyectos-con-buildout
[250] http://www.cursosgis.com/principales-errores-al-comenzar-con-python/
[251] https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos
[252] http://docs.python.org.ar/tutorial/2/classes.html

- Programación Orientada a Objetos (POO) - Scipy lecture notes[253].
- What's the meaning of underscores (_ & __) in Python variable names?[254].
- What is the meaning of a single and a double underscore before an object name?[255].
- Ver la Figura 4.12 anexo sobre *Clases y Objetos*.

B.10 Lección 10 - Decoradores y la librería estándar

B.10.1 Iteradores

- Ver el uso de *comprensión de listas* (página 266).
- Ver la Figura 4.14 anexo sobre *Generadores*.
- Ver la Figura 4.15 anexo sobre *Decoradores*.

B.10.2 Listas de comprensión

- Listas por comprensión - Scipy lecture notes[256].
- Ver la Figura 4.13 anexo sobre *Comprensión de Listas*.

[253] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/oop.html
[254] https://www.youtube.com/watch?v=ALZmCy2u0jQ
[255] https://stackoverflow.com/questions/1301346/what-is-the-meaning-of-a-single-and-a-double-underscore-before-an-object-name
[256] https://claudiovz.github.io/scipy-lecture-notes-ES/intro/language/control_flow.html#listas-por-comprension

APÉNDICE C

Operadores

A continuación, se ofrecen una referencia corta de los diversos *operadores* en Python:

C.1 Operadores de asignaciones

Una corta referencia de los *operadores de asignación* se ofrece a continuación:

Operador	Descripción	Ejemplo
=	asigna valor a una variable	`>>> r = 5` `>>> r1 = r`

+=	suma el valor a la variable	`>>> r = 5`
		`>>> r += 10; r`
		`15`

-=	resta el valor a la variable	`>>> r = 5`
		`>>> r -= 10; r`
		`-5`

*=	multiplica el valor a la variable	`>>> r = 5`
		`>>> r *= 10; r`
		`50`

/=	divide el valor a la variable	`>>> r = 5`
		`>>> r /= 10; r`
		`0`

**=	calcula el exponente del valor de la variable	`>>> r = 5`
		`>>> r **= 10; r`
		`9765625`

//=	calcula la división entera del valor de la variable	`>>> r = 5`
		`>>> r //= 10; r`
		`0`

%=	devuelve el resto de la división del valor de la variable	`>>> r = 5`
		`>>> r %= 10; r`
		`5`

Ver también:

Usted puede consultar para más información la explicación de cada uno de los *operadores de asignaciones* (página 32).

C.2 Operadores aritméticos

Una corta referencia de los *operadores aritméticos* se ofrece a continuación:

Operador	Descripción	Ejemplo
+	Suma	`>>> 3 + 2` `5`
-	Resta	`>>> 4 - 7` `-3`
-	Negación	`>>> -7` `-7`
*	Multiplicación	`>>> 2 * 6` `12`
**	Exponente	`>>> 2 ** 6` `64`
/	División	`>>> 3.5 / 2` `1.75`
//	División entera	`>>> 3.5 // 2` `1.0`
%	Módulo	`>>> 7 % 2` `1`

Ver también:

Usted puede consultar para más información la explicación de cada uno de los *operadores aritméticos* (página 35).

C.3 Operadores relacionales

Una corta referencia de los *operadores relacionales* se ofrece a continuación:

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	`>>> 5 == 3` `False`
!=	¿son distintos a y b?	`>>> 5 != 3` `True`
<	¿es a menor que b?	`>>> 5 < 3` `False`
>	¿es a mayor que b?	`>>> 5 > 3` `True`
<=	¿es a menor o igual que b?	`>>> 5 <= 5` `True`
>=	¿es a mayor o igual que b?	`>>> 5 >= 3` `True`

Ver también:

Usted puede consultar para más información la explicación de cada uno de los *operadores relacionales* (página 38).

C.4 Operadores lógicos

Una corta referencia de los *operadores lógicos* se ofrece a continuación:

Operador	Descripción	Ejemplo
and	¿se cumple a y b?	`>>> True and False` `False`
or	¿se cumple a o b?	`>>> True or False` `True`
not	No al valor	`>>> not True` `False`

Ver también:

Usted puede consultar para más información la explicación de cada uno de los *operadores lógicos* (página 86).

APÉNDICE D

Anexos del entrenamiento

A continuación varios material multimedia anexos a las lecciones del entrenamiento:

Figura 4.1: Vídeo Tutorial Python 1 - Introducción al Lenguaje de Programación[257], cortesía de CodigoFacilito.com[258].

Figura 4.2: Vídeo Tutorial Python 2 - Instalación[259], cortesía de CodigoFacilito.com[260].

[257] https://www.youtube.com/watch?v=CjmzDHMHxwU
[258] https://codigofacilito.com/
[259] https://www.youtube.com/watch?v=VTykmP-a2KY
[260] https://codigofacilito.com/
[261] https://www.youtube.com/watch?v=OtJEj7N9T6k
[262] https://codigofacilito.com/
[263] https://www.youtube.com/watch?v=ssnkfbBbcuw
[264] https://codigofacilito.com/

Figura 4.3: Vídeo Tutorial Python 3 - Hola Mundo[261], cortesía de CodigoFacilito.com[262].

Figura 4.4: Vídeo Tutorial Python 4 - Enteros, reales y operadores aritméticos[263], cortesía de CodigoFacilito.com[264].

Figura 4.5: Vídeo Tutorial Python 5 - Booleanos, operadores lógicos y cadenas[265], cortesía de CodigoFacilito.com[266].

Figura 4.6: Vídeo Tutorial Python 10 - Sentencias condicionales[267], cortesía de CodigoFacilito.com[268].

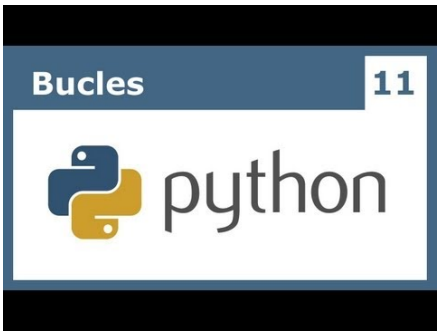

Figura 4.7: Vídeo Tutorial Python 11 - Bucles[269], cortesía de CodigoFacilito.com[270].

Figura 4.8: Vídeo Tutorial Python 12 - Funciones[271], cortesía de CodigoFacilito.com[272].

[265] https://www.youtube.com/watch?v=ZrxcqbFYjiw
[266] https://codigofacilito.com/
[267] https://www.youtube.com/watch?v=hLqKvB7tGWk
[268] https://codigofacilito.com/
[269] https://www.youtube.com/watch?v=IyI2ZuOq_xQ
[270] https://codigofacilito.com/
[271] https://www.youtube.com/watch?v=_C7Uj7O5o_Q
[272] https://codigofacilito.com/
[273] https://www.youtube.com/watch?v=N4NtB4r28h0
[274] https://www.youtube.com/watch?v=AzeUCuMvW6I
[275] https://codigofacilito.com/
[276] https://www.youtube.com/watch?v=B-JPXgxK3Oc
[277] https://codigofacilito.com/
[278] https://www.youtube.com/watch?v=VYXdpjCZojA
[279] https://codigofacilito.com/
[280] https://www.youtube.com/watch?v=87s8XQbUv1k
[281] https://codigofacilito.com/
[282] https://www.youtube.com/watch?v=tvHbC_OZV14
[283] https://codigofacilito.com/
[284] https://www.youtube.com/watch?v=TaIWx9paNIA
[285] https://codigofacilito.com/

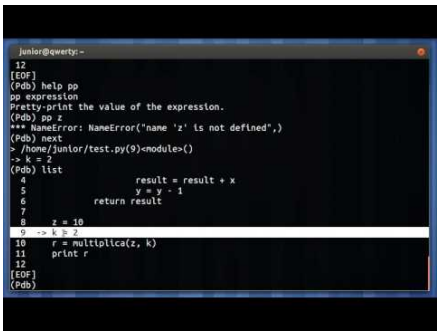

Figura 4.9: Vídeo Depurando um programa Python com pdb - Python Debugger[273], cortesía de *Youtube*.

Figura 4.10: Vídeo Tutorial Python 30 - Entrada Estándar rawInput[274], cortesía de CodigoFacilito.com[275].

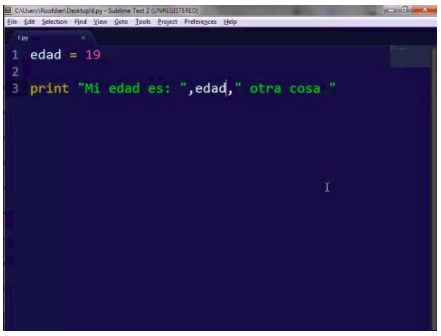

Figura 4.11: Vídeo Tutorial Python 31 - Salida Estándar rawInput[276], cortesía de CodigoFacilito.com[277].

Figura 4.12: Vídeo Tutorial Python 13 - Clases y Objetos[278], cortesía de CodigoFacilito.com[279].

Figura 4.13: Vídeo Tutorial Python 25 - Comprensión de Listas[280], cortesía de CodigoFacilito.com[281].

Figura 4.14: Vídeo Tutorial Python 26 - Generadores[282], cortesía de CodigoFacilito.com[283].

Figura 4.15: Vídeo Tutorial Python 27 - Decoradores[284], cortesía de CodigoFacilito.com[285].

APÉNDICE E

Glosario

Autor(es) Leonardo J. Caballero G.

Correo(s) leonardoc@plone.org

Compatible con Python 2.x, Python 3.x

Fecha 11 de Enero de 2021

A continuación una serie de términos usados en las tecnologías Python/Zope/Plone.

buildout En la herramienta buildout[286], es un conjunto de partes que describe como ensamblar una aplicación.

bundle Ver *Paquete bundle*.

Catalog Sinónimo en Ingles del termino *Catálogo*.

Catálogo Es un índice interno de los contenidos dentro de Plone para que se pueda buscar. El objetivo del catálogo es que sea accesible a través de la ZMI[287] a través de la herramienta portal_catalog[288].

Cheese shop Ver *PyPI*.

Collective Es un repositorio de código comunitario, para Productos Plone y productos de terceros, y es un sitio muy útil para buscar la ultima versión de código fuente del producto para cientos de productos de terceros a Plone. Los desarrolladores de nuevos productos de Plone son animados a compartir su código a través de Collective para que otros puedan encontrarlo, usarlo, y contribuir con correcciones / mejoras.

En la actualidad la comunidad ofrece dos repositorio Collective un basado en **Git** y otro **Subversion**.

Si usted quiere publicar un nuevo producto en el repositorio *Git de Collective* de Plone necesita obtener acceso de escritura[289] y seguir las reglas en github/collective, también puede consultarlo en la cuenta en github.com[290].

Si usted quiere publicar un nuevo producto en el repositorio *Subversion de Collective* de Plone necesita obtener acceso de escritura al repositorio[291] y crear su estructura básica de repositorio[292] para su producto, también puede consultarlo vía Web consulte el siguiente enlace[293].

[286] https://plone-spanish-docs.readthedocs.io/es/latest/buildout/replicacion_proyectos_python.html
[287] https://plone-spanish-docs.readthedocs.io/es/latest/zope/zmi/index.html
[288] https://plone-spanish-docs.readthedocs.io/es/latest/zope/zmi/index.html#portal-catalog
[289] https://collective.github.io/
[290] https://github.com/collective
[291] https://old.plone.org/countries/conosur/documentacion/obtener-acceso-de-escritura-al-repositorio-svn-de-plone
[292] https://old.plone.org/countries/conosur/documentacion/crear-un-nuevo-proyecto-en-el-repositorio-collective-de-plone
[293] https://svn.plone.org/svn/collective/

Declaración ZCML El uso concreto de una *Directiva ZCML* dentro de un archivo *ZCML*.

Directiva ZCML Una «etiqueta» *ZCML* como `<include />` o `<utility />`.

Egg Ver *paquetes Egg*.

esqueleto Los archivos y carpetas recreados por un usuario el cual los genero ejecutando alguna plantilla `templer` (`PasteScript`).

estructura 1) Una clase Python la cual controla la generación de un árbol de carpetas que contiene archivos.

2) Una unidad de carpetas y archivos proveídos por el sistema `templer` para ser usado en una plantilla o plantillas. Las estructuras proporcionan recursos estáticos compartidos, que pueden ser utilizados por cualquier paquete en el sistema de `templer`.

Las estructuras diferencian de las plantillas en que no proporcionan las *vars*.

filesystem Termino ingles File system, referido al sistema de archivo del sistema operativo.

Flujo de trabajo Es una forma muy poderosa de imitar los procesos de negocio de su organización, es también la forma en se manejan la configuración de seguridad de Plone.

Flujo de trabajos Plural del termino *Flujo de trabajo*.

grok Ver la documentacion del proyecto grok[294].

Instalación de Zope El software propio del servidor de aplicaciones.

Instancia de Zope Un directorio específico que contiene una configuración completa del servidor Zope.

local command Una clase Paste[295] la cual provee funcionalidad adicional a una estructura de esqueleto de proyecto que ha sido generada.

módulo Del Ingles `module`, es un archivo fuente Python; un archivo en el sistema de archivo que típicamente finaliza con la extensión `.py` o `.pyc`. Los modules son parte de un *paquete*.

Nombre de puntos Python Es la representación Python del «camino» para un determinado objeto / módulo / función, por ejemplo, `Products.GenericSetup.tool.exportToolset`. A menudo se utiliza como referencia en configuraciones `Paste` y `setuptools` a cosas en Python.

paquete Ver *Paquete Python*.

Paquete bundle Este paquete consististe en un archivo comprimido con todos los módulos que son necesario compilar o instalar en el *PYTHONPATH* de tu interprete `Python`.

paquete Egg Es una forma de empaquetar y distribuir paquetes Python. Cada Egg contiene un archivo `setup.py` con metadata (como el nombre del autor y la correo electrónico y información sobre el licenciamiento), como las dependencias del paquete.

La herramienta del *setuptools* <*que_es_setuptools*>, es la librería Python que permite usar el mecanismo de paquetes egg, esta es capaz de encontrar y descargar automáticamente las dependencias de los paquetes Egg que se instale.

Incluso es posible que dos paquetes Egg diferentes necesiten utilizar simultáneamente diferentes versiones de la misma dependencia. El formato de paquetes Eggs también soportan una función llamada `entry points`, una especie de mecanismo genérico de plug-in. Mucha más detalle sobre este tema se encuentra disponible en el sitio web de PEAK[296].

Paquete Python Es un termino generalmente usando para describir un módulo Python. en el más básico nivel, un paquete es un directorio que contiene un archivo `__init__.py` y algún código Python.

paquetes Egg Plural del termino *paquete Egg*.

Paquetes Python Plural del termino *Paquete Python*.

part En la herramienta *buildout*, es un conjunto opciones que le permite a usted construir una pieza de la aplicación.

[294] https://grok-community-docs.readthedocs.io/en/latest/
[295] https://paste.readthedocs.io/en/latest/
[296] http://peak.telecommunity.com/DevCenter/setuptools

plantilla 1) Una clase Python la cual controla la generación de un esqueleto. Las plantillas contiene una lista de variables para obtener la respuesta de un usuario. Las plantillas son ejecutadas con el comando `templer` suministrando el nombre de la plantilla como un argumento `templer basic_namespace my.package`.

2) Los archivos y carpetas proveídas un paquete `templer` como contenido a ser generado. Las respuestas proporcionadas por un usuario en respuesta a las variables se utilizan para rellenar los marcadores de posición en este contenido.

Producto Es una terminología usada por la comunidad Zope / Plone asociada a cualquier implementación de módulos / complementos y agregados que amplíen la funcionalidad por defecto que ofrece Zope / Plone. También son conocidos como *«Productos de terceros»* del Ingles Third-Party Products[297].

Producto Plone Es un tipo especial de paquete Zope usado para extender las funcionalidades de Plone. Se puede decir que son productos que su ámbito de uso es solo en el desde la interfaz gráfica de Plone.

Producto Zope Es un tipo especial de paquete Python usado para extender Zope. En las antiguas versiones de Zope, todos los productos eran carpetas que se ubican dentro de una carpeta especial llamada `Products` de una instancia Zope; estos tendrían un nombre de módulo Python que empiezan por «**Products.**». Por ejemplo, el núcleo de Plone es un producto llamado `CMFPlone`, conocido en Python como Products.CMFPlone[298].

Este tipo de productos esta disponibles desde la interfaz administrativa de Zope (ZMI)[299] de su instalación[300] donde deben acceder con las credenciales del usuario Administrador de Zope. Muchas veces el producto simplemente no hay que instalarlo por que se agregar automáticamente.

Productos Plural del termino *Producto*.

Productos Plone Plural del termino *Producto Plone*.

Productos Zope Plural del termino *Producto Zope*.

profile Una configuración «predeterminada» de un sitio, que se define en el sistema de archivos o en un archivo tar.

PyPI Siglas del termino en Ingles *Python Package Index*, es el servidor central de *paquetes Egg* Python ubicado en la dirección https://pypi.org/.

Python Package Index Ver *PyPI*.

PYTHONPATH Una lista de nombre de directorios, que contiene librerías Python, con la misma sintaxis como la declarativa `PATH` del shell del sistema operativo.

recipe En la herramienta *buildout*, es el software usado para crear partes de una instalación basada en sus opciones. Más información consulte el articulo Recipes Buildout[301].

setup.py El archivo `setup.py` es un módulo de Python, que por lo general indica que el módulo / paquete que está a punto de instalar ha sido empacado y distribuidos con `Distutils`, que es el estándar para la distribución de módulos de Python.

Con esto le permite instalar fácilmente paquetes de Python, a menudo es suficiente para escribir:

```
python setup.py install
```

Entonces el módulo Python se instalará.

Ver también:

- https://docs.python.org/2/install/index.html

[297] https://docs.plone.org/develop/addons/
[298] https://pypi.org/project/Products.CMFPlone
[299] https://plone-spanish-docs.readthedocs.io/es/latest/zope/zmi/index.html
[300] http://localhost:8080/manage
[301] https://plone-spanish-docs.readthedocs.io/es/latest/buildout/recipes.html

Temas / Apariencias Por lo general si un producto de Tema esta bien diseñado y implementado debe aplicarse de una ves al momento de instalarlo. En caso que no se aplique de una puede acceder a la sección Configuración de Temas[302] y cambiar el **Tema predeterminado** por el de su gusto.

Tipos de contenidos Los tipos de contenidos son productos que extienden la funcionalidad de **Agregar elemento** que permite agregar nuevos tipos de registros (Contenidos) a tu sitio. Esto quiere decir que si instala un tipo de contenido exitosamente debería poder acceder a usarlo desde el menú de **Agregar elemento** en el sitio Plone. Opcionalmente algunos productos instalan un panel de control del producto que puede acceder a este en la sección Configuración de Productos Adicionales[303].

var Diminutivo en singular del termino *variable*.

variable 1) Una pregunta que debe ser respondida por el usuario cuando esta generando una estructura de esqueleto de proyecto usando el sistema de plantilla `templer`. En este caso una variable (var) es una descripción de la información requerida, texto de ayuda y reglas de validación para garantizar la entrada de usuario correcta.

2) Una declarativa cuyo valor puede ser variable o constante dentro de un programa Python o en el sistema operativo.

variables Plural del termino *variable*.

vars Diminutivo en plural del termino *variable*.

Workflow Ver *Flujo de trabajo*.

ZCA

Zope Component Architecture La arquitectura de componentes de Zope (alias ZCA)[304], es un sistema que permite la aplicación y la expedición enchufabilidad complejo basado en objetos que implementan una interfaz.

ZCatalog Ver *Catalog*.

ZCML Siglas del termino en Ingles *Zope Configuration Mark-up Language*.

ZCML-slug Los así llamados «ZCML-slugs», era configuraciones que estaban destinados a enlazar dentro de un directorio una configuración especial en una instalación de Zope, por lo general se ven como `collective.foo-configure.zcml`. Estas configuraciones ya no están más en uso, pueden ser eliminados agregando las configuraciones del paquete z3c.autoinclude[305].

Zope Configuration Mark-up Language Es un dialecto XML utilizado por Zope para las tareas de configuración. ZCML es capaz de realizar diferentes tipos de declaración de configuración. Es utilizado para extender y conectar a los sistemas basados en la *Zope Component Architecture*.

`Zope 3` tiene la política de separar el código actual y moverlo a los archivos de configuración independientes, típicamente un archivo `configure.zcml` en un buildout. Este archivo configura la instancia Zope. El concepto "Configuración" podría ser un poco engañoso aquí y debe ser pensado o tomarse más cableado.

`ZCML`, el lenguaje de configuración basado en `XML` que se utiliza para esto, se adapta a hacer el registro de componentes y declaraciones de seguridad, en su mayor parte. Al habilitar o deshabilitar ciertos componentes en ZCML, puede configurar ciertas políticas de la aplicación general. En `Zope 2`, habilitar y deshabilitar componentes significa eliminar o remover un determinado producto `Zope 2`. Cuando está ahí, se importa y se carga automáticamente. Este no es el caso en `Zope 3` Si no habilita explícitamente, no va a ser encontrado.

El *grok* proyecto ha adoptado un enfoque diferente para el mismo problema, y permite el registro de componentes, etc haciendo declarativa de código Python. Ambos enfoques son posibles en Plone.

[302] http://localhost:8080/Plone/@@skins-controlpanel
[303] http://localhost:8080/Plone/prefs_install_products_form
[304] https://plone-spanish-docs.readthedocs.io/es/latest/programacion/zca/zca-es.html
[305] https://pypi.org/project/z3c.autoinclude

APÉNDICE F

Licenciamientos

F.1 Reconocimiento-CompartirIgual 3.0 Venezuela de Creative Commons

Sobre esta licencia

Esta documentación se distribuye bajo los términos de la licencia Reconocimiento-CompartirIgual 3.0 Venezuela de Creative Commons[306].

Usted es libre de:

- Compartir — copiar y redistribuir el material en cualquier medio o formato.
- Adaptar — remezclar, transformar y crear a partir del material.
- Para cualquier propósito, incluso comercialmente.
- El licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia.

Bajo los siguientes términos:

- Reconocimiento - Usted debe dar el crédito apropiado, proporcionar un enlace a la licencia, y de indicar si se han realizado cambios. Usted puede hacerlo de cualquier manera razonable, pero no en una manera que sugiere el licenciante a usted o que apruebe su utilización.
- CompartirIgual - Si usted mezcla, transforma o crea nuevo material a partir de esta obra, usted podrá distribuir su contribución siempre que utilice la misma licencia que la obra original.

Términos detallados de la licencia: https://creativecommons.org/licenses/by-sa/3.0/ve/legalcode

[306] https://creativecommons.org/licenses/by-sa/3.0/ve/

Índice