Curso Python

Eduardo Espinosa Avila

2024

Contents

1 Introducción.

1.1 Acerca de Python.

1.1.1 ¿Qué es Python?

Python es un lenguaje de programación multiparadigma; esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, programación estructurada y programación funcional.

Python fue desarrollado por Guido van Rossum en 1991.

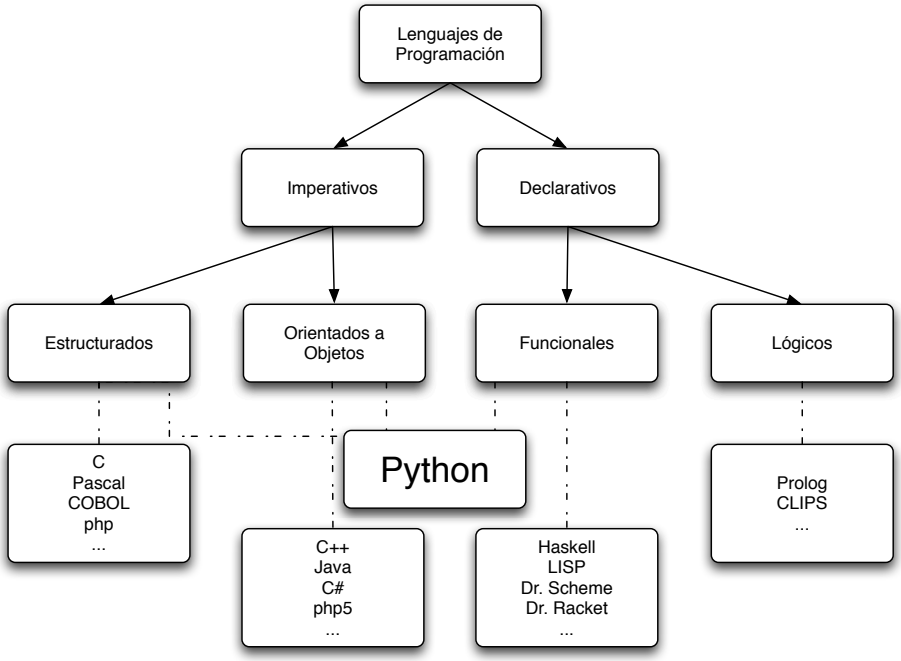

Figure 1.1: Ubicación de Python en la clasificación de acuerdo al paradigma

Algunas veces se hace referencia a Python como un *pseudocódigo ejecutable* (http://www.melbpc.org.au/pcupdate/2108/ veamos esto con un ejemplo sencillo.

Supongamos la siguiente definición matemática para obtener el factorial de un número natural: $\prod_{i=1}^{n} i$. Esta misma forma del factorial puede definirse como: $\forall i \in (1, n).x \leftarrow x * i$. Un posible pseudocódigo que realice este cálculo es el siguiente:

```
Para cada elemento en el rango (1, n): hacer x <- x*i
```

El código Python que se encarga de realizar lo anterior es el siguiente:

```
x = 1
for i in range(1,5): x*=i
```

Que, en realidad es la descripción en idioma inglés del pseudocódigo mostrado anteriormente.

1.1.2 Ventajas.

Python es bueno para muchas situaciones:

Fácil de Usar

Programadores familiarizados con *lenguajes tradicionales* encontrarán relativamente fácil aprender Python. Todas las expresiones usuales tales como ciclos, condicionales, arreglos, etc., están incluídas, pero muchas son más sencillas en Python. Algunas razones son:

- ◇ *Los tipos se asocian a objetos, no a varibles.* Una variable puede contener valores de cualquier tipo, y una lista puede contener objetos de tipos diferentes. Esto también implica que el *casting* de tipos no es necesario y tampoco lo es predeclarar variables.
- ◇ *Típicamente Python opera en un muy alto nivel de abstracción.* En parte por la construcción del mismo lenguaje y en parte gracias a la biblioteca estándar tan extensa que esta incluída con la distribución de Python. Un programa que descargue una página web se puede escribir con dos o tres líneas de código.
- ◇ *Las reglas sintácticas son muy sencillas.* Aún los principiantes pueden escribir código útil rápidamente.

Python es muy conveniente para desarrollo rápido de aplicaciones.

Expresividad.

Python es un lenguaje muy expresivo, en este contexto, expresivo significa que una línea puede hacer más que una línea en la casi cualquier otro lenguaje. Las ventajas son obvias: menos líneas de código requieren menor tiempo para escribirlas, aún más, menos líneas de código impican mayor facilidad para mantener y depurar programas.

Por ejemplo, supóngase que se requiere intercambiar un valor entre dos variables; el siguiente código lo realizaría en un lenguaje tipo C o Java:

```
int temp = var1;
var1 = var2;
var2 = temp;
```

Por el contrario, el mismo intercambio de valores se puede hacer con la siguiente línea de código Python:

```
var2, var1 = var1, var2
```

Además de la simplicidad del código, debe notarse que no es necesario hacer uso (explícito) de una variable auxiliar para realizar el intercambio.

Legibilidad.

Otra ventaja de Python, es su fácilidad de lectura; podría pensarse que un programa será leído solo por una computadora, pero también será leído por seres humanos, ya sea al depurar, mantener o modificar el código; en cualquier caso, entre más legible sea, será más sencillo realizarlo.

"Baterías incluídas"

Otra ventaja de Python es su filosofía de "baterías incuídas" cuando se habla de bibliotecas. La idea es que al instalar Python se debería tener todo lo necesario para hacer trabajo real. Por esto la biblioteca estándar de Python incluye módulos para manejo de *email*, páginas *web*, bases de datos, llamadas al sistema operativo, desarrollo de GUIs y más. Por ejemplo, se puede escribir un servidor *web* para compartir archivos de un directorio con solo dos líneas de código:

```
import http.server
http.server.test(HandlerClass=http.server.SimpleHTTPRequestHandler)
```

No hay necesidad de instalar bibliotecas extras para controlar conexiones ni HTTP; ya están incluídas en Python.

Multiplataforma.

Python también es un excelente lenguaje multiplataforma; corre en *Windows*, *Mac*, *Linux*, *UNIX*. Dado que es interpretado, el mismo código puede ejecutarse en cualquier plataforma que tenga instalado un intérprete de Python. Incluso existen versiones de Python que corren en Java (Jython) y .NET (IronPython), incrementando las plataformas de ejecución de Python.

Open Source.

Python está desarrollado con el modelo *open source* y está disponible libremente, se puede descargar e instalar cualquier versión, además puede usarse para desarrollar *software* comercial o aplicaciones personales sin necesidad de pagar por él.

Greg Stein, ingeniero administrador del grupo Open Source de Google, dijo en su presentación en la SDForum Python Meeting:

En Google, Python es uno de 3 "lenguajes oficiales", junto con C++ y Java. Oficial significa que los empleados de Google pueden usar estos lenguajes en proyectos de producción. Internamente, la gente de Google puede usar muchas otras tecnologías, incluyendo PHP, C#, Ruby y Perl. Python está bien adecuado a los procesos de ingeniería en Google. El típico proyecto en Google tiene un equipo pequeño (de 3 personas) y una corta duración (de 3 meses). Después de que el proyecto se ha terminado, los desarrolladores pueden irse a otros proyectos. Los proyectos más grandes pueden subdividirse en otros más pequeños presentables en 3 meses, y los equipos pueden elegir su propio lenguaje para el proyecto.

1.1.3 Desventajas

Python tiene ventajas, aún así ningún lenguaje puede hacer todo, por tanto Python no es la solución perfecta para todas las necesidades. Para decidir si Python es el lenguaje correcto para una situación específica, es necesario considerar las áreas en las que Python no es tan bueno:

No es el lenguaje más rápido.

Un posible inconveniente de Python es su velocidad de ejecución, es un lenguaje interpretado, debido a esto, la mayoría de las veces Python resulta en programas que se ejecutan más lento que un programa realizado en C. Sin embargo, las computadoras actuales tienen el poder suficiente para que en la gran mayoría de las aplicaciones la velocidad del programa no sea tan importante como la rapidez de desarrollo. Además, Python puede extenderse con módulos escritos en C, que ejecuten porciones de código intensivo para el CPU.

No posee las bibliotecas más extensas.

Python posee una excelente colección de bibliotecas desde la instalación y muchas otras están disponibles, sin embargo, lenguajes como C, Java o Perl tienen colecciones más extensas disponibles y en algunas ocasiones ofrecen varias soluciones cuando Python tiene solamente una. Sin embargo, Python puede extenderse con bibliotecas de otro lenguajes. Para resolver los problemas más comunes de computación, el soporte de las bibliotecas estándar de Python es excelente.

No tiene revisión de tipos.

Contrario a otros lenguajes, las variables en Python son más *etiquetas* que hacen referencia a varios objetos: enteros, cadenas, clases, etc.; esto implica que aún cuando estos objetos tienen tipos, las variables que hacen referencia a ellos no están ligados a un tipo en particular. Es posible que alguna variable x refiera a una cadena en una línea y a un entero en otra.

El hecho de que Python asocia tipos con objetos y no con variables, quiere decir que el intérprete no ayudará a encontrar "*errores*" en el tipo de una variable.

1.1.4 ¿Quién lo usa?

En general, Python, goza de una gran base de usuarios, y una muy activa comunidad de desarrolladores. Dado que Python ha existido por casi 20 años y ha sido ampliamente utilizado, también es muy estable y robusto. Además de ser utilizado por usuarios individuales, Python también se aplica a productos reales en empresas reales. Por ejemplo:

- ◇ *Google* hace un amplio uso de Python en su sistema de búsqueda web, y emplea al creador de Python.
- ◇ El servicio de video *YouTube*, es en gran medida escrito en Python.
- ◇ El popular peer-to-peer *BitTorrent* para compartir archivos es un programa Python.
- ◇ *Intel*, *Cisco*, *Hewlett-Packard*, *Seagate*, *Qualcomm* e *IBM* utilizan Python para las pruebas de hardware.
- ◇ *Industrial Light & Magic*, *Pixar*, y otros utilizan Python en la producción de cine de animación.
- ◇ *JPMorgan Chase*, *Getco*, y *Citadel* aplican Python para los mercados financieros de previsión.
- ◇ La *NASA*, Los *Alamos*, *Fermilab*, *JPL*, y otros utilizan Python para tareas de programación científica.
- ◇ *iRobot* utiliza Python para desarrollar aspiradoras robóticas comerciales.
- ◇ *ESRI* utiliza Python como una herramienta de personalización de usuario final para sus populares productos de cartografía *GIS*.
- ◇ La *NSA* utiliza Python para criptografía y análisis de inteligencia.

- ◇ El servidor de correo electrónico de *IronPort* utiliza más de 1 millón de líneas de código Python para hacer su trabajo.
- ◇ El proyecto *One Laptop Per Child* (OLPC) basa su interfaz de usuario y el modelo de actividad en Python.
- ◇ El servidor web *Zope* y el *CMS Plone* están escritos completamente en Python.
- ◇ **EL LENGUAJE FAVORITO PARA ML, AI, etc.**

Y la lista continua, para más detalles sobre empresas que utilizan hoy Python, véase el sitio web de Python para mayor referencia https://www.python.org/about/success/.

1.2 Primeros pasos.

1.2.1 Instalación.

Instalar Python es un asunto sencillo, sin importar el sistema operativo que se utilice. El primer paso es obtener una distribución para el equipo usado, para esto se debe acceder a la sección de descargas es www.python.org y seguir las instrucciones referentes al sistema operativo utilizado.

1.2.2 Modo interactivo.

El modo interactivo permite ejecutar diversos comandos de Python; para iniciarlo, en una terminal (interfaz de comandos en *windows*) se debe escribir *python* en la línea de comandos. Si existen varias versiones instaladas, puede especificarse la deseada, por ejemplo *python3*.

La mayoría de las plataformas tienen un mecanismo de *historial de comandos*, se pueden recuperar instucciones escritas con ayuda de las teclas de dirección.

Para salir del modo interactivo, se debe usar la combinación de teclas CTRL+D (CTRL+Z en *windows*).

1.2.3 Hola mundo.

Una vez en el modo interactivo, el *prompt* está identificado por >>>. Este es el *prompt* de comandos de Python e indica que se puede escribir un comando para ser ejecutado o una expresión para ser evaluada. Comenzaremos con el obligatorio "*Hola mundo*", el cual consiste de una sola línea en Python:

```
print ("Hola mundo")
 Hola mundo
```

1.2.4 El Zen de Python

Los usuarios de Python se refieren a menudo a la **Filosofía Python** que es bastante análoga a la Filosofía de Unix.

El código que sigue los principios de Python de legibilidad y transparencia se dice que es ***pythonico***. Contrariamente, el código opaco u ofuscado es bautizado como ***no pythonico*** (*unpythonic* en inglés).

Estos principios fueron famosamente descritos por el desarrollador de Python Tim Peters en el **Zen de Python**.

◇ 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.

◇ Aunque lo práctico gana a la pureza.

◇ 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!

En cualquier momento se puede consultar esta guía desde el modo interactivo de Python:

```
import this
```

También es interesante probar:

```
import antigravity
```

1.2.5 Usando el modo interactivo para explorar Python.

Existen algunas herramientas útiles para explorar Python:

La primera es la función *help()* que tiene dos modos. Escribir solo *help()* en el *prompt* para entrar al sistema de ayuda. Una vez dentro, el prompt cambia a *help>*, y se puede ingresar el nombre de algún módulo, por ejemplo *math* y explorar la documentación de ese tema.

Usualmente es más conveniente utilizar *help*() de forma más dirigida; ingresando un tipo o una variable como parámetro de *help*(), se obtiene ayuda inmediata de la documentación de ese tipo:

```
x = 5
help(x)
```

Usar *help()* de esta forma es más práctico para revisar la sintaxis de un método o el comportamiento de un objeto.

Otra función útil es *dir()*, la cual lista los objetos en un *espacio* particular. Si se usa sin parámetros, lista las variables globales, pero también puede listar los objetos de un módulo o incluso de un tipo:

```
dir()
dir(int)
```

La función *type*() devuelve el tipo de dato del objeto que recibe como parámetro:

```
type(5)
type(1+1j)
type(None)
```

Además se pueden usar otras dos funciones para ver los valores de variables locales y globales respectivamente: *locals()* y *globals()*.

1.2.6 Otros recursos

- ◇ Se puede consultar la documentación oficial en cualquier momento en la URL http://docs.python.org
- ◇ Así mismo, se puede revisar la lista de libros gratuitos en línea:
  https://github.com/vhf/free-programming-books/blob/master/free-programming-books.md#python
- ◇ Esta presentación muestra algunas diferencias entre Python 2 y 3:
  https://www.dropbox.com/s/83ppa5iykqmr14z/Py2v3Hackers2013.pptx

2 Tipos de datos básicos.

2.1 *None.*

None es una constante especial de Python, cuyo valor es **nulo**.

- ◇ *None* **no** es lo mismo que *False*.
- ◇ *None* **no** es 0.
- ◇ *None* **no** es una cadena vacía.
- ◇ Si comparas *None* con cualquier otra cosa que no tenga valor *None* obtendrás siempre *False*.
- ◇ *None* es el único valor nulo.

Tiene su propio tipo de dato (*NoneType*), se puede asignar *None* a cualquier variable y todas las variables cuyo valor es *None* son iguales entre sí.

```
x = None
type(x)
type(None)
```

2.2 *Booleanos.*

Los *booleanos* son verdaderos o falsos. Python tiene dos constantes, de ingenioso nombre *True* y *False*, que se pueden usar para asignar valores booleanos de manera directa. Las expresiones también pueden evaluarse como valores *booleanos*.

En ciertos lugares (como las sentencias *if*), Python espera que la evaluación de expresiones produzca un *booleano*. Estos lugares se denominan contextos *booleanos*. En uno de estos contextos se puede usar prácticamente cualquier expresión y Python tratará de determinar su valor de verdad.

2.3 Números.

Python incluye manejo de números para enteros, de punto flotante y complejos. Pueden realizarse operaciones con los operadores aritméticos: + (suma), - (resta), * (multiplicación), / división, ** (exponenciación) y % (módulo).

Con enteros:

```
x = 2*3+1-5
x = 5/2
x = 5%2
x = 5**2
```

Pero todos funcionan también con flotantes:

```
x = 2.1*3.2+1.5-5.3
x = 5.0/2
x = 5/2.0
x = 5.3%2.1
x = 4.5**2.3
```

Y con complejos (sin necesidad de importar alguna biblioteca):

```
x = 2.1+3.2j
y = 1.5+5.3j
z = x + y
z = x - y
z = x * y
z = x / y
z = x ** y
```

En Python 3, NO está disponible la operación módulo entre complejos; en Python 2 sí:

```
z = x % y
```

2.4 Listas.

Cuando se habla de una *lista*, puede pensarse en "un arreglo cuyo tamaño hay que declarar con antelación, que sólo puede contener elementos del mismo tipo". Pero en realidad una lista de Python es mucho más poderosa.

2.4.1 Creación de una lista.

Crear una lista es sencillo, solo se deben usar corchetes para encerrar una lista valores separados por comas:

```
x = [1, 2, 3] # una lista de enteros
x = ["uno", "dos", "tres"] # una lista de cadenas
```

Pero una lista puede contener valores de diferentes tipos:

```
x = [5, "dos", [1, 2, 3]] # una lista de tipos diferentes: entero, cadena
    , lista
```

La función *len()* devuelve el número de elementos que tiene una lista.

```
x = [5, "dos", [1, 2, 3]]
len(x)
```

2.4.2 Índices.

Entender la forma en la que trabajan los índices de listas es muy útil.

Se pueden obtener los elementos de un lista de Python usando notación similar a C o Java; al igual que en C y otros lenguajes, el índice inicial para Python es 0 y devuelve el primer elemento de la lista:

```
x = ["uno", "dos", "tres", "cuatro"]
x[0]
x[3]
```

Pero el indexamiento de Python es más flexible, si se utiliza un índice negativo, asume que se busca una posición determinada contando desde el final de la lista, con -1 la última posición:

```
x = ["uno", "dos", "tres", "cuatro"]
x[-1]
x[-3]
```

2.4.3 Particionado de una lista.

Una vez definida una lista, se puede obtener cualquier parte suya en forma de una nueva lista. A esto se le denomina particionado (*slicing*) de la lista.

```
x = ["uno", "dos", "tres", "cuatro"]
x[0:3]
x[1:-1]
x[1:-3]
```

2.4.4 Modificado de una lista.

Se puede utilizar el indexamiento para **obtener un valor** y también para modificarlo.

```
x = [1, 2, 3, 4]
x[1] = "dos"
```

Agregar un elemento es una operación común, para esto Python incluye un método especial:

```
x = [1, 2, 3, 4]
x.append("cinco")
```

A veces se necesita **concatenar** una lista al final de otra:

```
x = [1, 2, 3, 4]
y = [5, 6, 7]
x.append(y)
```

Pero en este caso y es un elemento de x, no es el resultado deseado; para **concatenar** se utiliza otro método:

```
x = [1, 2, 3, 4]
y = [5, 6, 7]
x.extend(y)
```

También es posible **insertar** un elemento en una posición determinada:

```
x = [1, 2, 3, 4]
x.insert(2, "centro")
x.insert(0, "inicio")
x.insert(-1, "negativo")
```

El comando más usado para **eliminar** elementos de una lista es *del*:

```
x = [1, 2, 3, 4, 5, 6, 7, 8]
del x[1]
del x[:2]
del x[2:]
```

El método *remove* **elimina** la primera incidencia del valor dado:

```
x = [1, 2, 3, 4, 5, 6, 7, 8]
x.remove(3)
```

Para **invertir** una lista, se tiene un método especializado:

```
x = [1, 2, 3, 4, 5, 6, 7, 8]
x.reverse()
```

2.4.5 Ordenamiento.

Python el método *sort* para **ordenar** listas:

```
x = [3, 8, 4, 0, 2, 1]
x.sort()
```

Este método modifica la lista original, para evitarlo, se puede trabajar sobre una copia:

```
x = [3, 8, 4, 0, 2, 1]
y = x[:]
y.sort()
```

También sirve para ordenar cadenas y listas:

```
x = ["uno","dos","tres","cuatro"]
x.sort()
x = [[3, 5], [2, 9], [2, 3], [4, 1], [3, 2]]
x.sort()
```

¿Complejos?

```
x = [1+1j,2-3j,-2+2j,5+9j] # ¿Se pueden ordenar complejos?
x.sort() # NO
```

¿Por qué? → http://goo.gl/UVYsR

2.4.6 Otras operaciones comunes

Para determinar si un elemento **pertenece o no** a una lista:

```
x = [1, 2, 3, 4, 5, 6, 7, 8]
3 in x
3 not in x
```

Otra forma de **concatenar** listas es hacer uso del operador +:

```
x = [1, 2, 3, 4]
y = [5, 6, 7]
z = x + y
```

Inicialización con el operador *:

```
x = [None] * 4
y = [1, 2] * 3
```

También se puede buscar el **mínimo** y **máximo** de una lista:

```
x = [3, 8, 4, 0, 2, 1]
min(x)
max(x)
```

Con cadenas:

```
x = ["uno","dos","tres","cuatro"]
min(x)
max(x)
```

Se puede **buscar el índice** de algún elemento:

```
x = [3, 8, 4, 0, 2, 1]
x.index(4)
```

Y podemos **obtener el número de apariciones** de un elemento en una lista:

```
x = [1, 2, 2, 3, 5, 2, 5]
x.count(2)
x.count(0)
```

Finalmente, Python incluye la función *pop*, que **obtiene el último elemento** de una lista y lo elimina de la misma:

```
x = [3, 8, 4, 0, 2, 1]
x.pop() # devuelve 1
```

Y tiene una variante que obtiene el valor del índice indicado y también lo elimina:

```
x = [3, 8, 4, 0, 2, 1]
x.pop(2) # devuelve 4
```

2.5 Tuplas.

Las tuplas son estructuras de datos similares a las listas, pero no pueden modificarse, solo pueden crearse. Una tupla es una lista inmutable.

2.5.1 Creación de una tupla.

Una tupla se define de la misma manera que una lista, excepto que el conjunto de elementos se encierra entre paréntesis en lugar de corchetes.

- ◇ Los elementos de una tupla tienen un orden definido, al igual que en las listas.
- ◇ Los índices de las tuplas empiezan en 0, exactamente como las listas.
- ◇ Los índices negativos empiezan a contar desde el final de la tupla, como con las listas.
- ◇ También funciona el particionado, igual que en las listas.

Muchas funciones de listas sirven para manipular tuplas:

```
x = ('a','b','c') # una tupla de tres elementos
x[0]
x[1:]
len(x)
min(x)
max(x)
'c' in x
'd' not in x
```

La principal diferencia entre tuplas y listas, es que una tupla no puede modificarse:

```
x = ('a','b','c')
x[2] = 'd'
```

Pero se pueden crear tuplas con los operadores + y *:

```
x = ('a','b','c')
y = x + x
y = x * 4
```

La copia de una tupla se hace de la misma forma que las listas:

```
x = ('a','b','c')
y = x[:]
```

Una pregunta interesante es por qué Python incluye tuplas si son tan similares a las listas, y de hecho más limitadas, ya que son estáticas, las razones son las siguientes:

- ◇ Las tuplas son más rápidas que las listas. Si se requiere un conjunto constante de valores para iterar sobre él, una tupla es más conveniente que una lista.
- ◇ El código es más seguro si se "protegen contra escritura" los datos que no haga falta cambiar.
- ◇ Algunas tuplas se pueden usar como claves para diccionarios, como se verá más adelante, mientras que las listas no se pueden usar para este fin.

2.5.2 Conversiones entre listas y tuplas.

Las tuplas pueden convertirse en listas con la función *list*; de manera similar, las listas pueden convertirse en tuplas con la función *tuple*:

```
x = list((1, 2, 3, 4))
x = tuple([1, 2, 3, 4])
```

Como nota interesante, una lista es una forma conveniente de separar una cadena en caracteres:

```
x = list("Hola")
```

Esto funciona debido a que *list* (y *tuple*) se aplican a cualquier secuencia de Python, y una cadena es solo una secuencia de caracteres.

2.6 Conjuntos.

Un conjunto es una colección desordenada de objetos, usado en situaciones en las que la pertenencia y la unicidad de cada objeto es importante.

2.6.1 Creación de un conjunto.

Para crear conjuntos en Python, se utiliza la función *set*, que recibe una lista o una tupla como parámetro:

```
x = set([1, 2, 3, 4])
x = set((1, 2, 3, 4))
```

2.6.2 Operaciones con conjuntos.

Python ofrece varias operaciones específicas de conjuntos:

```
x = set([1, 2, 3, 4])
x.add(5)
x.add(5) # un conjunto no almacena elementos repetidos
x.remove(4) # elimina un elemento
x.discard(4) # también sirve para eliminar
2 in x
4 in x
```

Además, si se tienen dos conjuntos puede obtenerse la unión, intersección y diferencia entre ellos:

```
x = set([1, 2, 3, 4])
y = set([3, 4, 5, 6])
z = x | y # unión
z = x & y # intersección
z = x - y # diferencia
z = x ^ y # diferencia simétrica
```

2.7 Cadenas.

El procesamiento de cadenas (*strings*) es una de las fortalezas de Python.

2.7.1 Creación de cadenas.

Existen varias formas de delimitar cadenas en Python:

```
x = "Cadena con comillas dobles, puede contener 'comillas simples'"
x = 'Cadena con comillas simples, puede contener "comillas dobles"'
x = '''\tCadena que inicia con un tabulador y una línea nueva\n'''
x = """Cadena con triple comillas dobles,
 puede contener líneas nuevas 'reales'"""
```

Se pueden concatenar cadenas con el operador +:

```
x = "Hola"
y = ' mundo'
z = x + y
```

Y también existe el operador *multiplicación* de cadenas *:

```
x = "x" * 10
```

2.7.2 Métodos asociados a cadenas.

La mayoría de los métodos para cadenas de Python están integrados a la clase *string*, por tanto cualquier cadena los tiene asociados; para hacer uso de algún método se utiliza el operador punto (.).

Cómo las cadenas son inmutables, los métodos devuelven un resultado, pero no modifican la cadena original.

Los métodos *split* y *join*.

Son dos métodos muy útiles cuando se trabaja con cadenas, la función *split* devuelve una lista de subcadenas de la cadena original:

```
x = "Hola mundo"
y = x.split()
```

Se puede especificar una cadena usada como separador:

```
x = "Hola&&buen&&día"
y = x.split('&&')
```

Si se especifican n particiones, *split* generará $n+1$ *subcadenas*:

```
x = "a b c d e"
y = x.split(' ', 1)
y = x.split(' ', 3)
y = x.split(' ', 9)
```

La función *join* permite concatenar cadenas, pero de una mejor forma que el operador +:

```
x = " ".join(['join','concatena','espacios'])
x = ":".join(['join','concatena','lo','que','sea'])
x = "".join(['incluso','cadena','vacía'])
```

Convirtiendo cadenas a números.

Las funciones *int* y *float* se utilizan para conviertir cadenas en números enteros o de punto flotante, respectivamente:

```
x = int('253')
x = float('128.345')
x = int('12.5') # error, un entero no tiene punto decimal
x = float('xx.yy') # error, no puede representarse como número
```

Además *int* puede recibir un segundo argumento:

```
x = int('1000',8)
x = int('101',2)
x = int('ff',16)
x = int('12345',5)
```

Python cuenta con métodos para cambiar mayúsculas a minúsculas y viceversa *upper*, *lower*, *title*; y otras para revisar algunas características de una cadena: *isdigit*, *isalpha*, *isupper*, *islower*.

Búsquedas.

Las cadenas proveen varios métodos para realizar búsquedas simples en cadenas.

Existen cuatro métodos principales para hacer búsquedas en cadenas Python: *find*, *rfind*, *index* y *rindex*, muy similares entre ellas. Además el método *count* obtiene el número de apariciones de una cadena dentro de otra.

La función *find* requiere un argumento, la cadena a buscar y devuelve la posición del primer caracter en la primera aparición, si no se encuentra, devuelve -1:

```
x = "Ferrocarril"
x.find('rr')
x.find('y')
```

Además, *find* tiene dos argumentos opcionaes, ambos enteros, si se especifica el primero, la búsqueda se realiza a partir de esa posición en la cadena, si se utiliza el segundo argumento, la búsqueda se hace en la subcadena limitada por ambos argumentos:

```
x = "Ferrocarril"
x.find('rr',4)
x.find('rr',4,6)
```

Otro par de funciones de búsqueda útiles son *startswith* y *endswith*, realizan una búsqueda al inicio y al final de la cadena, respectivamente, y devuelven un valor *booleano*:

```
x = "Ferrocarril"
x.startswith('fer')
x.endswith('rril')
```

Pero en Python 3 tiene una ventaja, se pueden buscar varias cadenas, si se especifican como una tupla:

```
x = "Ferrocarril"
x.endswith(('rril',"il","l"))
```

Conviertiendo objetos a cadenas.

Prácticamente cualquier cosa en Python puede representarse como cadena, usando la función *repr*:

```
x = [1, 2, 3, 4]
"La lista x es : " + repr(x)
x = (1, 2, 3, 4)
"La tupla x es : " + repr(x)
x = set([1, 2, 3, 4])
"El conjunto x es : " + repr(x)
```

La función *repr* puede utilizarse para obtener la representación de cadena de casi cualquier objeto Python:

```
repr(len)
```

2.7.3 Formateando cadenas.

Recordemos que se pueden concatenar cadenas con el operador + y el método *join*, sin embargo con estas opciones no es posible dar formato alguno a las cadenas.

Para dar formato a cadenas, Python posee el operador % (*string modulus*), se utiliza para combinar valores con cadenas:

```
"El %s es la %s de los %s" % ("tequila", "bebida", "dioses")
```

El operador % (al centro, no las %s de la cadena izquierda), requiere dos partes: del lado izquierdo una cadena y del lado derecho una tupla. El operador % revisa la cadena buscando *secuencias de formato* y produce una nueva cadena sustituyendo las secuencias de formato con los valores de la tupla en el mismo orden.

La tupla puede contener valores que no son cadenas:

```
"La %s contiene %s" % ("lista", [5, "dos", [1, 2, 3]])
```

Secuencias de formato.

Existen diferentes secuencias de formato:

```
"Entero : %d" % (2)
"Flotante : %f" % (5.2481)
"Flotante : %6.2f" % (5.2481) # 6 posiciones 2 de ellas para la parte
    decimal
"Flotante : %.3e" % (0.00000192) # notación científica con 3 cifras
    decimales
```

Python a partir de la versión 3 provee además de % el método *format* para dar formato a cadenas.

```
"El {0} es la {1} de los {2}".format("tequila", "bebida", "dioses")
```

En este caso, se tienen tres campos reemplazables {0}, {1} y {2}, los cuales se rellenan con el primero, segundo y tercer parámetro respectivamente, la diferencia y ventaja respecto a %, es que se puede colocar el campo {0} en cualquier posición y será reemplazado siempre por el primer parámetro.

El método format también reconoce parámetros con nombres para reemplazar campos:

```
"El {bebida} es la bebida de los {usuarios}".format(bebida="tequila",
    usuarios="dioses")
```

También es posible utilizar ambas formas de reemplazo de parámetros, e incluso se puede acceder a atributos y elementos de esos parámetros:

```
"El {0} es la bebida de los {usuarios[1]}".format("tequila", usuarios=["
    hombres","dioses","otros"])
```

Lectura de valores por teclado.

Python proporciona varias formas de leer valores por teclado, la más sencilla es la función *input*:

```
x = input("Ingresa algo : ")
print(x)
```

La función lee lo que se escribe como cadena, y se debe procesar para utilizarla como algún otro tipo.

2.8 Diccionarios.

Un diccionario es un conjunto desordenado de pares *clave-valor* (*keys-values*), similares a los *hashtables*, o los *arreglos asociativos* de otros lenguajes. Cuando se añade una clave a un diccionario, también se debe añadir su valor asociado (que puede cambiarse después). Los diccionarios de Python están optimizados para recuperar un valor a partir de la clave, pero no al contrario.

2.8.1 Creación de diccionarios.

Crear un diccionario es sencillo, existen dos formas para hacerlo, la primera es crear un diccionario vacío solamente con llaves de apertura y cierre:

```
x = {}
```

Y después añadir parejas clave-valor:

```
x['red']='rojo'
x['green']='verde'
x['blue']='azul'
```

Aunque tambien se puede crear especificand pares clave : valor separados por comas:

```
x = {'red':'rojo', 'green':'verde', 'blue':'azul'}
```

Las claves y los valores pueden ser de tipos distintos:

```
x = {"uno":1, 2:"dos", 2.5:"flotante", 1+1j:"complejo", "lista":[1,2,3],
    (0,1):"tupla"}
```

Nota: Las claves del diccionario son únicas.

2.8.2 Modificando un diccionario.

Los diccionarios no tienen ningún límite de tamaño predefinido. Se pueden añadir pares clave-valor en cualquier momento, o modificar el valor de una clave ya asignada.

```
x = {'red':'rojo', 'green':'verde', 'blue':'azul'}
x["white"]="negro"
x["white"]="blanco"
```

2.8.3 Otras operaciones con diccionarios.

Se puede obtener el número de elementos de un diccionario, así como preguntar si una clave está en el mismo (pero no se puede preguntar por el valor):

```
x = {'red':'rojo', 'green':'verde', 'blue':'azul'}
len(x)
"red" in x
"rojo" in x
```

Pero se pueden obtener las claves o los valores como una lista:

```
x = {'red':'rojo', 'green':'verde', 'blue':'azul'}
y = x.keys()
y = x.values()
```

Y ahora podemos preguntar por claves y valores:

```
x = {'red':'rojo', 'green':'verde', 'blue':'azul'}
y = x.keys()
"red" in y
y = x.values()
"rojo" in y
```

También se pueden obtener los pares clave-valor del diccionario como una lista de tuplas:

```
x = {'red':'rojo', 'green':'verde', 'blue':'azul'}
x.items()
```

Otros métodos útiles son *get*, *setdefault* y *copy*:

```
x = {'red':'rojo', 'green':'verde', 'blue':'azul'}
x.get("green","no disponible")
x.get("black","no disponible")
x.setdefault("green","no disponible")
x.setdefault("black","no disponible")
y = x.copy()
```

2.8.4 Formateando cadenas con diccionarios.

Se puede combinar el uso del operador % para formatear cadenas con los valores de un diccionario, utilizando las claves del diccionario como *nombres* en una secuencia de formato:

```
x = {'e': 2.718, 'pi': 3.14159}
print("%(pi).2f \n %(pi).4f \n %(e).2f" % x)
```

2.8.5 Tipos válidos para usarse como claves.

Como se mostró anteriormente se pueden utilizar diferentes tipos de Python como claves de un diccionario, pero no todos: cualquier tipo Python que sea inmutable puede usarse como clave de un diccionario, lo anterior excluye las listas, conjuntos y los mismos diccionarios como posibles claves.

Ejemplo de uso de tuplas como claves.

Una matriz es una tabla de dos dimensiones como la siguiente:

$$\begin{bmatrix} 3 & 0 & 4 & 0 \\ 0 & 0 & 1 & 7 \\ 0 & 2 & 0 & 3 \\ 6 & 0 & 1 & 0 \end{bmatrix}$$

Una forma de representar esta matriz en Python es la siguiente:

```
m = [[3,0,4,0],[0,0,1,7],[0,2,0,3],[6,0,1,0]]
```

Y se puede tener acceso a los elementos de la matriz usando índices:

```
m[2][3] # devuelve 3
```

Sin embargo, en algunas aplicaciones es común tener matrices de miles de elementos, muchos de los cuales tienen valor 0; una forma de economizar memoria es hacer uso de matrices dispersas (*sparse matrices*).

En Python es simple implementar matrices dispersas, utilizando diccionarios con tuplas como claves, por ejemplo, la matriz anterior se puede representar como:

```
m = {(0,0):3, (0,2):4, (1,2):1, (1,3):7, (2,1):2, (2,3):3,
     (3,0):6, (3,2):1}
```

Y se puede utilizar el método *get* para acceder a los elementos:

```
m.get((2,3),0) # devuelve 3
m.get((0,1),0) # devuelve 0
```

3 Control de flujo.

Python tiene un conjunto completo de estructuras para controlar el flujo de la ejecución del programa. Un punto importante para el uso de sentencias de control de flujo, es que a diferencia de otros lenguajes, la indentación es obligatoria, con esto se evita el uso de marcadores de bloque.

3.1 Sentencia if-elif-else.

En esta sentencia, se ejecuta el bloque de código que se encuentra después de la primera condición verdadera (del *if* o algún *elif*), si ninguna condición es verdadera, se ejecuta el bloque de código después de la sentencia *else*. La estructura más general es:

```
if cond1:
        bloque1
elif cond2:
        bloque2
. . .
elif condN:
        bloqueN
else:
        bloqueElse
```

Un ejemplo sencillo es el siguiente:

```
x = input("Ingresa un número : ")
y = input("Ingresa otro número : ")
if x<y:
        print("y es mayor")
 elif x>y:
        print("x es mayor")
 else:
        print("son iguales")
```

Nota: al trabajar en la línea de comandos, si se da enter en una línea de código incompleta aparecen de forma automática los puntos suspensivos (. . .), esperando el resto de la instrucción y para terminarla, se debe dar *enter* en una línea vacía.

3.2 Ciclo while.

El ciclo *while* ejecuta un bloque de código mientras la condición se evalua a *True*, una vez que se evalua a *False*, se ejecuta el bloque de código que sigue al cierre de indentación:

```
while cond:
        bloqueWhile
bloqueSiguiente
```

Por ejemplo:

```
x,y,z=0,5,0
while x < y:
        z += x
        x += 1
```

3.3 Ciclo for.

En Python el ciclo *for* es diferente a la mayoría de los otros lenguajes de programación, este ciclo itera sobre cualquier objeto iterable (cualquier objeto que contenga secuencias de valores); por ejemplo listas, tuplas, cadenas ó diccionarios. La sintaxis es la siguiente:

```
for x in secuencia:
        bloqueFor
bloqueSiguiente
```

Por ejemplo:

```
l1,l2=[1,2,3,1+1j],[]
for x in l1:
        l2.append(x/1.1)
```

3.3.1 La función *range*.

Algunas veces es necesario iterar con índices explícitos (posiciones en una lista). La función *range* junto con la función *len* generan una secuencia de índices que pueden usarse en el ciclo; como se ve en el siguiente ejemplo:

```
l=[1,2-3j,3,1+1j,4]
for i in range(len(l)):
        if l[i].imag is not 0:
                print("Hay un complejo en la posición %d" % i)
```

Si la función range recibe un sólo argumento n, regresa una secuencia de 0 a n-1; además, *range* puede utilizarse para generar una secuencias de valores en un rango:

```
list(range(3,12)) # list se utiliza para que se vea
list(range(6,2)) # el resultado pero no es necesaria
```

Y también puede recibir otro argumento, que indica el valor del incremento a aplicar (puede ser negativo):

```
list(range(3,12,2))
list(range(6,2,-1))
```

Las secuencias generadas por *range* siempre incluyen el valor inicial y nunca el valor final.

3.4 Sentencias *break* y *continue*.

Tanto el ciclo *while* como el *for* tienen una sintaxis más general a la presentada anteriormente:

```
while cond:
        bloqueWhile
else:
        bloqueElse
```

```
for x in secuencia:
        bloqueFor
else:
        bloqueElse
```

En ambos casos, una vez que el ciclo termina, se ejecuta el bloque de la sentencia *else*; la diferencia es que si dentro del bloque correspondiente al ciclo se encuentra una sentencia *break*, el ciclo termina y no se ejecuta el código de *else*. Por otro lado, si el ciclo tiene una sentencia *continue*, se termina la iteración actual y se continua con la siguiente iteración.

Ejemplo:

```
import random
x=random.randint(1,10)
for i in range(3):
   y=int(input('adivina el número : '))
   if x>y:
     print('para arriba')
   elif x<y:
     print('para abajo')
   else:
     print('adivinaste!')
     break
   continue
   print('No llega a esta línea')
 else:
   print('No adivinaste, el número es : '+repr(x))
```

3.5 Valores *booleanos* y expresiones.

Los ejemplos anteriores de control de flujo, verifican el valor condicional con comparaciones, pero no muestran de manera explícita que es un valor *True* o *False* en Python, o que expresiones se pueden utilizar como condición.

3.5.1 La mayoría de los objetos en Python pueden usarse en pruebas *booleanas*.

Python incluye el tipo *Boolean* que puede almacenar los valores *True* ó *False*. Toda expresión *booleana* devuelve *True* ó *False*; las expresiones *booleanas* se forman con operadores de comparación ($<$, $>$, $==$, $!=$, $<=$, $>=$, $<>$) y con operadores lógicos (*and*, *or*, *not*, *is*, *is not*).

Adicionalmente, Python utiliza las siguientes reglas cuando un objeto se utiliza como condición:

- ◇ Cualquiera de los siguientes valores son *False*:
  - Los números 0, 0.0 y $0 + 0j$
  - La cadena vacía “”
  - La lista vacía []
  - El conjunto vacío *set*()
  - El diccionario vacío {}
  - El valor *None*
- ◇ Y cualquier valor diferente a los anteriores, es *True*.

Existen otras estructuras que no se han revisado, pero se aplica la misma regla general. Algunos objetos como no tienen una definición de 0, vacío ó nulo (como es el caso de los archivos) y no deben usarse en un contexto booleano.

3.6 Listas y diccionarios por "*entendimiento/completitud*".

Utilizar un ciclo *for* para iterar sobre una lista para modificar ó seleccionar elementos indivuales y, a partir de ellos crear una nueva lista o diccionario es muy común; un ciclo como tal se vería de la siguiente forma:

```
l = [1, 2, 3, 4]
l_2 = []
for x in l:
        l_2.append(x ** 2)
```

Esto es tan común que Python posee una forma especial para estas operaciones, llamado "*entendimiento*" (*comprehension*). Una lista o diccionario por entendimiento es un ciclo de una sola línea que crea una nueva lista o diccionario a partir de otra lista.

La sintaxis general es la siguiente:

```
listaNueva = [expr1 for var in lista if expr2]
diccionarioNuevo = {expr1:expr2 for var in lista if expr3}
```

Por ejemplo, el código siguiente hace exactamente lo mismo que el ejemplo inicial, pero usando entendimiento:

```
l = [1, 2, 3, 4]
l_2 = [x ** 2 for x in l]
```

Incluso puede usarse una expresión *if* para elegir ciertos elementos de la lista:

```
l = range(20)
l_2 = [x**2 for x in l if x%2 is 0]
```

Además en Python 3 se pueden generar diccionarios por entendimiento. Un diccionario por entendimiento es similar, pero es necesario proporcionar pares clave-valor; por ejemplo, si se desea un diccionario con clave un número y valor el cuadrado de cada número, se puede obtener por entendimiento, de la siguiente manera:

```
l = range(20)
dic = {x : x ** 2 for x in l if x%2 is 1}
```

La creación de listas y diccionarios por entendimiento es un mecanismo muy útil y poderoso que puede hacer más sencillo el procesamiento de listas.

4 Funciones.

4.1 Definición de funciones.

La definición de funciones en Python tiene la siguiente sintaxis:

```
def nombre_funcion(param1, param2, . . .):
        cuerpo
```

Como en las estructuras de control, Python utiliza indentación para delimitar el cuerpo de la función, por ejemplo:

```
def suma(n):
        """Realiza la suma de n a 0"""
        s = 0
        while n > 0:
                s += n
                n -= 1
        return s
```

La línea 2 es opcional y es una *cadena de documentación* (*docstring*) y se puede recuperar su valor con *suma.__doc__*. Usualmente la cadena de documentación se utiliza para dar una sinopsis de la función en varias líneas.

Ahora está disponible en el ambiente el uso de la función *suma*:

```
suma(5)
x = suma(5)
```

4.2 Opciones para los parámetros de una función.

La mayoría de las funciones necesitan parámetro, cada lenguaje tiene reglas específicas para el manejo de parámetros. Python es flexible y provee tres opciones para definir parámetros de funciones.

4.2.1 Paso de parámetros por posición.

La forma más sencilla de pasar parámetros a un función en Python es por posición. En la primer línea de la función se especifican los nombres de las variables para cada parámetro; cuando se llama la función, los parámetros usados en el llamado se asocian con los parámetros, basados en el orden.

El siguiente ejemplo calcula x elevado a la potencia y:

```
def pow(x, y):
        """Calcula x ^ y"""
        p = 1
        while y > 0:
                p *= x
                y -= 1
        return p
```

Esta función requiere que el número de parámetros sea exactamente el mismo al número de parámetros de la definición de la función, o generará una excepción de *TypeError*:

```
pow(5, 2) # devuelve 25
pow(5) # genera una excepción TypeError
```

4.2.2 Valores de parámetros por omisión.

Los parámetros de una función pueden tener valores por omisión (*default*), que se declaran asignando una valor en la primera línea de la definición de la función, la sintaxis es:

```
def funcion(param1=val1, param2=val2, param3=val3, . . .):
        cuerpo
```

El siguiente ejemplo también realiza el cálculo de x a la potencia y, pero si no se especifica un valor para y, se utiliza el valor por omisión 2 y la función obtiene el cuadrado del primer parámetro:

```
def pow(x, y=2):
        """Calcula x ^ y"""
        p = 1
        while y > 0:
                p *= x
                y -= 1
        return p
```

4.2.3 Paso de parámetros por nombre del parámetro.

Se pueden pasar argumentos a una función usando el nombre del parámetro correspondiente, en lugar de su posición. Usando la función anterior, la función puede llamarse como:

```
pow(5, 3) # devuelve 125
pow(5) # devuelve 25
pow(y=3, x=5) # devuelve 125
```

Dado que en el útimo llamado los argumentos tienen el nombre de los parámetros, la posición es irrelevante. Este mecanismo se conoce como *paso por palabra clave* (*keyword passing*).

Combinar el paso de parámetros por palabra clave junto con los valores por omisión puede resultar muy útil cuande se desea definir funciones con un gran número de argumentos posibles.

4.2.4 Número variable de parámetros.

Las funciones en Python pueden tener un número variable de argumentos, esto puede hacerse de dos formas diferentes. La primera sirve para el caso en que no se conoce el número de argumentos, y se desea capturar estos argumentos en una lista. El segundo método puede capturar un número arbitrario de argumentos pasados por palabra clave, que no tienen un nombre de parámetro que corresponda con la definición de la función, en un diccionario.

Manipulación de un número indefinido de argumentos por posición.

Colocar un * antes del nombre del último parámetrode una función provoca que los argumentos "extras" que no usen palabra clave en la llamada a la función, sean captados como una tupla en el parámetro dado.

El siguiente ejemplo obtiene el máximo elemento de una lista de elementos:

```
def maximo(*numeros):
        if not numeros:
                return None
        numMax = numeros[0]
        for n in numeros[1:]:
                if n > numMax: numMax = n
        return numMax
```

Y se puede llamar de la siguiente manera:

```
maximo() # devuelve None
maximo(1,5,2,6,9,7,4,8,3)
```

Manipulación de un número indefinido de argumentos por palabra clave.

Un número arbitrario de argumentos por palabra clave también puede ser manipulado, si se antepone ** al último parámetro de la función, esté colectará los argumentos "extras" que sean pasados por palabra clave en un diccionario. La clave de cada entrada será la palabra clave (nombre del parámetro) y el valor será el mismo que se pasa como argumento.

Por ejemplo:

```
def ejemDic(x, y, **dic):
        print("x = %s y = %s dict = %s" % (x, y, dic))
```

Y el llamado es de la siguiente manera:

```
ejemDic(y = 1, x = 2)
ejemDic(y = 1, x = 2, z = 4, w = 6)
```

5 Módulos.

Los módulos se utilizan para organizar proyectos grandes de Python, por ejemplos, la biblioteca estándar está dividida en módulos para que su manejo sea más sencillo. No es necesario organizar todos

los códigos de Python en módulos, pero si un programa ocupa muchas líneas de código, o si se desea reutilizar código, es buena idea hacerlo de esta forma.

Un módulo es un archivo que contiene código en el que se definen un conjunto de funciones de Python y otros objetos. El nombre del módulo se deriva del nombre del archivo y se puede importar con la sentencia:

```
import NombreModulo
```

Y para utilizar algún *nombre* definido dentro del módulo, se realiza:

```
NombreModulo.nombre
```

También existe una forma más general para importar nombres de objetos:

```
from NombreModulo import *
```

El * reemplaza los nombres declarados en el módulo; esto importa todos los *nombres públicos* del módulo (aquellos que no inician con guión bajo) y los hace disponibles sin necesidad de escribir el nombre del módulo antes del nombre del objeto.

Se debe tener cuidado al utilizar esta forma de importar objetos: si dos módulos definen un mismo nombre de objeto y se importan ambos módulos usando esta forma de importación, se presenta un *choque* de nombres, y el del segundo módulo reemplazará al del primero; además esto hará más difícil determinar donde se originan los nombres que se están utilizando en el código.

Es mucho mejor importar nombres simples, en ocasiones con un alias si existe posibilidad de nombres repetidos:

```
from NombreModulo import nombre [as alias]
```

5.1 Bibliotecas y módulos de terceros.

Al inicio de la sección se indica que la biblioteca estándar de Python se encuentra dividida en módulos para que sea más manejable; una vez instalado, toda esta funcionalidad está disponible, lo único que debe hacerse es importar los módulos ó nombres explícitamente para poder usarlos. La biblioteca estándar incluye muchos módulos, éstos pueden consultarse en la referencia de bibliotecas de Python (*Python Library Reference*) en http://docs.python.org/library/.

Además, en el sitio *web* de Python (www.python.org), existen módulos desarrollados por terceros y ligas a ellos. Lo único que hay que hacer para utilizarlos es consultar en la documentación la forma de instalarlos y realizar el proceso que ahí se indique.

6 Clases y programación orientada a objetos.

Está sección presenta una breve introducción a las características de Programación Orientada a Objetos (POO) que se incluyen en Python.

6.1 Declaración de clases.

Una clase de Python es un *tipo de dato efectivo*; todos los tipos de datos nativos en Python son clases, y Python ofrece herramientas para manipular todos los aspectos del comportamiento de clases.

Una clase se declara utilizando la palabra reservada *class*:

```
class MiClase(object):
# todas las clases heredan de object
        cuerpo
```

El cuerpo es una serie de sentencias Python, usualmente asignaciones de variables y definiciones de funciones, pero no son necesarias, el cuerpo puede ser sola una sentencia *pass*.

Por convención, los identificadores de clases llevan mayúscula la primer letra y cada cambio de palabra. Una vez definida una clase, se puede crear un objeto (instancia) del tipo de esa clase, llamando al nombre de la clase como función:

```
objeto = MiClase()
```

Ejemplo:

```
class MiClase():
  """MiClase: Ejemplo 1"""
  pass
```

Prueba en la interfaz interactiva de Python:

```
obj = ej1.MiClase()
obj.__doc__
```

Por otro lado, la forma en la que se modelan sistemas basados en la ideología OO, utiliza diagramas del Lenguaje Unificado de Modelado, UML por sus siglas en inglés (*Unified Modeling Language*). El primer diagrama definido en este lenguaje es el diagrama de clases, su esquema general se muestra en la siguiente figura.

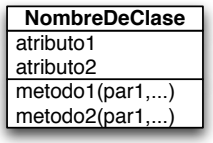

Figure 6.1: Esquema general del Diagrama de Clases.

Esta figura muestra la aplicación de algunas convenciones para diagramas de clases:

- ◇ El nombre de la clase inicia con mayúscula y las demás letras en minúsculas, salvo que haya cambio de palabra, en cuyo caso la primer letra será mayúscula.
- ◇ Los nombres de atributos y métodos son en minúsculas, salvo que haya cambio de palabra, en cuyo caso la primer letra será mayúscula.

6.2 Atributos.

En POO, se tiene el concepto de atributos (variables, *fields*), Python provee capacidad de tener atributos de instancia y atributos de clase.

6.2.1 Atributos de instancia.

Las variables de instancia son la característica más básica de la POO, un atributo de instancia es aquella que pertenece a cada objeto creado, y cuyo valor puede ser diferente a ese atributo en otras instancias.

Se pueden inicializar incluyendo la *función inicializadora* __*init*__ en el cuerpo de la clase, esta función se ejecuta siempre que se crea una nueva instancia de la clase:

```
class MiClase():
        def __init__(self, arg1, , argN):
                self.atr1 = arg1
                . . .
                self.atrN = argN
```

Por convención, *self* siempre es el nombre del primer argumento de la función __*init*__; *self* siempre hace referencia al objeto que llama al método, en este caso, al objeto que se acaba de crear.

Además, en Python se pueden crear atributos de instancia asignando un valor a atributo nuevo:

```
objeto.nuevoAtributo = valor
```

Siempre que se utiliza un atributo de instancia (tanto para asignación como para recuperar valor), requieren mención explícita el objeto que lo contiene: *objeto.atributo*. Por ejemplo:

```
class Circulo():
        def __init__(self):
                self.radio = 1
```

En una sesión interactiva, se puede probar con:

```
circ1 = Circulo()
print(circ1.radio)
circ2 = Circulo()
print(circ2.radio)
circ1.radio = 5
circ1.atNvo = 'hola' # creando un nuevo atributo
print(circ1.radio)
print(circ1.atNvo)
```

6.2.2 Atributos de clase.

Una variable de clase está asociada a la clase, no a un objeto particular, y todos los objetos de esa clase pueden acceder al valor

Para crear un atributo de clase, se asigna un valor en el cuerpo de la clase, no en la función *__init__*, una vez creada, puede ser vista por todas las instancias de la clase, por ejemplo:

```
class Circulo():
        pi = 3.14159
        def __init__(self):
                self.radio = 1
```

El diagrama de clases correspondiente es el de la siguiente figura, es importante notar que no se hace diferencia entre el atributo de instancia y el de clase (no es parte del lenguaje).

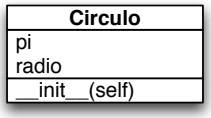

Figure 6.2: Diagrama de clases del ejemplo.

En el modo interactivo, se puede probar:

```
Circulo.pi
Circulo.pi = 5
Circulo.pi
```

Así es como se espera que funcione un atributo de clase, no hay necesidad de crear objetos del tipo de la clase para acceder al valor.

Sin embargo también es deseable poder obtener este valor a través de instancias creadas a partir de determinada clase, esto puede realizarse utilizando en atributo especial $\_\_class\_\_$, este atributo devuelve la clase a la cual pertenece determinada instancia:

```
Circulo.pi
circ1.__class__
circ1.__class__.pi
circ2.__class__.pi
circ1.__class__.pi = 5
circ2.__class__.pi
```

Una *rareza* en los atributos.

En Python sucede algo *raro* (en el sentido de que no sucede en otros lenguajes) si no se tiene cuidado al utilizar atributos de instancia y de clase; si se solicita un nombre de variable a un objeto, primero busca en los atributos inicializados en el método $\_\_init\_\_$, si no lo encuentra sube un nivel y busca en los atributos de clase, si no existe en este nivel, envía un error:

```
circ1.pi
```

Ahora, si por error se asigna un valor a este atributo, lo que hace es crear un nuevo atributo de instancia, no modifica el atributo de clase:

```
circ1.pi = 5 # crea un atributo de instancia
circ1.pi # imprime 5
circ1.__class__.pi # imprime 3.1415
```

6.3 Métodos.

Un método es una función asociada a una clase particular; por ejemplo, el método *__init__*, este método es llamado cuando se crea un nuevo objeto, en el siguiente ejemplo se declara otro método que devuelve el área del círculo:

```
class Circulo():
        pi = 3.14159
        def __init__(self):
                self.radio = 1
        def area(self):
                return self.__class__.pi * self.radio ** 2
```

Al igual que el método *__init__*, los métodos definidos llevan *self* como primer argumento, para probar este método, se hace lo siguiente:

```
circ1 = Circulo()
circ1.radio = 5
circ1.area()
```

Es deseable poder inicializar el valor de argumentos en la creación de un objeto, esto se puede hacer agregando argumentos al método *__init__*, el siguiente ejemplo recibe como argumento el radio del círculo y lo asigna en la creación, también es posible utilizar todas las técnicas de paso de argumentos vistas anteriormente:

```
class Circulo():
        pi = 3.14159
        def __init__(self, radio=1):
                self.radio = radio
        def area(self):
                return self.__class__.pi * self.radio ** 2
```

Ahora en la creación del objeto se puede indicar el radio del círculo:

```
circ1 = Circulo(5)
```

Las clases en Python, tienen métodos que corresponden con tales métodos en lenguajes como java; adicionalmente, Python tiene métodos de clase que son más avanzados.

6.3.1 Métodos estáticos.

Al igual que en java, se pueden invocar métodos estáticos sin necesidad de crear instancias de la clase, aunque pueden llamarse utilizando un objeto. Para crear un método estático, es necesario utilizar la sentencia @*staticmethod* antes del método:

```
class Circulo():
        todosLosCirculos = [] # lista que almacenará todos los objetos
        pi = 3.14159
        def __init__(self, radio=1):
                """Inicializa un Circulo con el radio dado"""
                self.radio = radio
                self.__class__.todosLosCirculos.append(self) # agrega el
                    objeto actual a la lista
        def area(self):
                """Obtiene el área del Circulo"""
                return self.__class__.pi * self.radio ** 2
        @staticmethod
        def areaTotal():
                """Obtiene el área de todos los Circulos"""
                total = 0
                for c in Circulo.todosLosCirculos:
                        total = total + c.area()
                return total
```

Se puede probar interactivamente:

```
c1=Circulo()
c2=Circulo(3)
Circulo.areaTotal()
c3=Circulo(5)
c2.areaTotal()
```

También se incluyen *docstrings* para los métodos y pueden verse con *__doc__*.

6.3.2 Métodos de clase.

Los métodos de clase son similares a los métodos estáticos, ambos pueden invocarse sin necesidad de crear objetos; pero los métodos de clase reciben implícitamente la clase a la que pertenecen como primer parámetro, con esto se puede codificar de forma más sencilla. Para crear un método de clase, se utiliza la sentencia @*classmethod* antes del método:

```
class Circulo():
        todosLosCirculos = [] # lista que almacenará todos los objetos
        pi = 3.14159
        def __init__(self, radio=1):
                """Inicializa un Circulo con el radio dado"""
                self.radio = radio
                self.__class__.todosLosCirculos.append(self) # agrega el
                    objeto actual a la lista
        def area(self):
                """Obtiene el área del Circulo"""
                return self.__class__.pi * self.radio ** 2
        @classmethod
        def areaTotal(cls):
                """Obtiene el área de todos los Circulos"""
                total = 0
                for c in cls.todosLosCirculos:
                        total = total + c.area()
                return total
```

El parámetro de métodos de clase, tradicionalmente es *cls*, con esto, se puede utilizar *cls* en lugar de *self.__class__*, o dejar "código duro" con referencia explícita a *Circulo*.

El diagrama de clases completo del ejemplo se muestra a continuación, tampoco se hace distinción entre métdos estáticos y de clase.

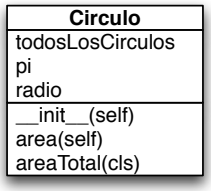

Figure 6.3: Diagrama de clases del ejemplo.

6.3.3 Verificar que un objeto sea del tipo deseado

La función type devuelve la clase a la cuál pertenece un objeto determinado; sin embargo, este valor no puede usarse para comprobar si es de algún tipo específica. Para realizar esto último, se suele utilzar el atributo "*__name__*", por ejemplo:

```
def checkType(valor, tipo):
  return type(valor).__name__ == tipo

checkType(5,'int')
checkType(5,'float')
checkType(5.,'float')
```

Esta sección es una introducción a la POO en Python con el objetivo de poder revisar y entender códigos que utilicen este paradigma.

7 ¿Qué sigue?

Python es muy útil para desarrollar aplicaciones de forma rápida, además de lo mostrado en este documento se pueden realizar muchas otras cosas como las nombradas a continuación. Un buen manual para introducción a algunas de ellas se encuentra en http://diveintopython.org/; además, como se mencionó anteriormente en http://docs.python.org/library/ se encuentra la documentación oficial de la *Python Library Reference*, en la cual se incluyen ejemplos de su uso.

Apredizaje Automático e Inteligencia Artificial.

En los últimos años el uso de Python se ha incrementado exponencialmente y es el lenguaje más utilizado para aplicaciones de ML & AI.

Paquetes.

Los paquetes permiten crear bibliotecas de código que puede estar esparcido en múltiples archivos y directorios, lo que permite tener mejor organización de grandes colecciones de código que usando un módulo.

Conceptos avanzados de POO.

Python posee muchas opciones relacionadas con orientación a objetos; añadiendo atributos especiales a los métodos, se pueden simular otras clases, incluso los tipos básicos; también es posible modificar su comportamiento según las necesidades propias; también es posible controlar la creación y el comportamiento de las clases, por medio de *meta-clases*.

Interfaces Gráficas de Usuario (GUI).

Python incluye un módulo muy completo llamado *Tkinter*, un *framework* que permite desarrollo rápido de aplicaciones gráficas interactivas, así como control de los aspectos de la interfaz.

Acceso a bases de datos

El acceso a bases de datos es una parte muy importante de muchas aplicaciones, incluyendo aplicaciones *web* dinámicas; utilizando módulos externos, Python puede comunicarse con las bases de datos comunes, la interfaz de todas ellas utilizan el estándar DB-API 2.0, lo cual facilita todas las operaciones sobre bases de datos.

Programación en red.

La biblioteca estándar de Python posee todo lo necesario para controlar los protocolos estándares de Internet, así como crear clientes y servidores; en muchos casos, puede crearse un servidor con unas pocas líneas de código.

Aplicaciones *web*.

Aunque el módulo *http.server* tiene todo lo básico para servicios *web*, se necesita más que sólo lo básico para crear aplicaciones completas; es necesario manejo de usuarios, autentificación, sesiones, etc.; y se necesita un mecanismo para generar páginas *html*. La solución es utilizar el *web framework*, con este se han creado varios *frameworks* en Python, llegando a proyectos de última generación, como *Zope* y *Plone*, *Django*, *TurboGears*, *web2py* y muchos otros.