¿Qué es Python?

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90 cuyo nombre está inspirado en el grupo de cómicos ingleses "Monty Python". Es un lenguaje similar a Perl, pero con una sintaxis muy limpia y que favorece un código legible. Se trata de un lenguaje interpretado o de script, con tipado dinámico, fuertemente tipado, multiplataforma y orientado a objetos.

Lenguaje interpretado o de script
Un lenguaje interpretado o de script es aquel que se ejecuta utilizando un programa intermedio llamado intérprete, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora (lenguajes compilados).

Fuertemente tipado
No se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente. Por ejemplo, si tenemos una variable que contiene un texto (variable de tipo cadena o string) no podremos tratarla como un número (sumar la cadena "9" y el número 8). En otros lenguajes el tipo de la variable cambiaría para adaptarse al comportamiento esperado, aunque esto es más propenso a errores.

Multiplataforma
El intérprete de Python está disponible en multitud de plataformas (UNIX, Solaris, Linux, DOS, Windows, OS/2, Mac OS, etc.) por lo que si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas sin grandes cambios.

Orientado a objetos
La orientación a objetos es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. La ejecución del programa consiste en una serie de interacciones entre los objetos. Python también permite la programación imperativa, programación funcional y programación orientada a aspectos.

Instalación de Python

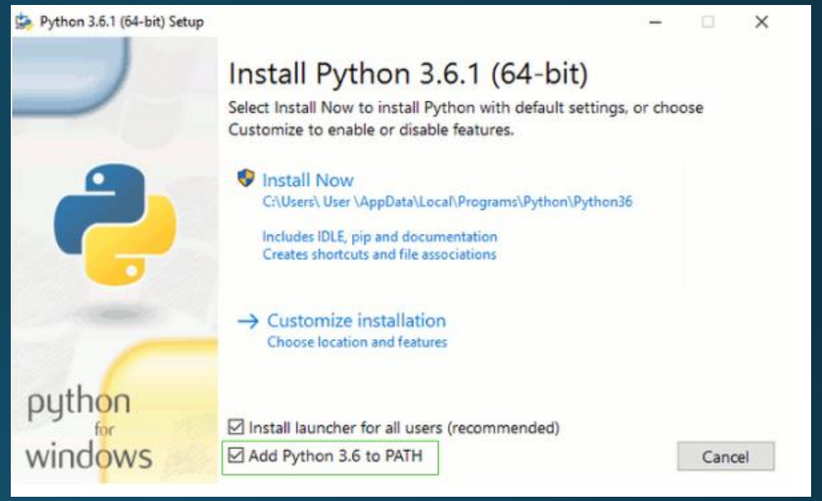
Primero comprueba si tu ordenador ejecuta la versión 32 bits de Windows o la de 64, en "Tipo de sistema" en la página de "Acerca de". Para llegar a esta página, intenta uno de estos métodos: Presiona la tecla de Windows y la tecla Pause/Break al mismo tiempo Abre el Panel de Control desde el menú de Windows, después accede a Sistema & y Seguridad, luego a Sistema Presiona el botón de Windows, luego accede a Configuración > Sistema > Acerca de

Puedes descargar Python para Windows desde la siguiente web

https://www.python.org/downloads/windows/

Da clic en el enlace "Latest Python 3 Release -Python x.x.x". Si tu ordenador ejecuta la versión de 64 bits de Windows, descarga Windows x86-64 executable installer. De lo contrario, descarga Windows x86 ejecutable installer. Después de descargar el instalador, deberías ejecutarlo (dándole doble click) y seguir las instrucciones.

Una cosa para tener en cuenta: Durante la instalación, verás una ventana de "Setup". Asegúrate de marcar las casillas "Add Python 3.6 to PATH" o "Add Python to your environment variables" y hacer click en "Install Now", como se muestra aquí (puede que se vea un poco diferente si estás instalando una versión diferente):



Cuando la instalación se complete, verás un cuadro de diálogo con un enlace que puedes seguir para saber más sobre Python o sobre la versión que has instalado. Cierra o cancela ese diálogo. Comprobar que se instalo Python con el siguiente comando desde la cmd python –versión

```
C:\Users\Raul>python --version
Python 3.10.2

C:\Users\Raul>
```

Números enteros y reales

Números enteros

Son aquellos números positivos o negativos que no tienen decimales (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 podemos almacenar números de -231 a 231 - 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. El tipo long de Python permite almacenar números de cualquier precisión, estando limitados solo 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.

Reales

Los números reales son los que tienen decimales. En Python se expresan mediante el tipo float. En otros lenguajes de programación, como C, tenemos 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 podemos representar van desde ±2,2250738585072020 x 10-308 hasta ±1,79769313486231570×10308. 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 contamos también con un nuevo tipo Decimal, 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 poder 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

Operadores

Operadores aritméticos

Veamos ahora qué podemos hacer con nuestros números usando los operadores por defecto. Para operaciones más complejas podemos recurrir al módulo math.

OPERADORES ARITMETICOS			
Operador	Descripción	Ejemplo	Resultado
+	Suma	c = 3 + 5	c = 8
-	Resta	c = 4 - 2	c = 2
-	Negación	c = -7	c = -7
*	Multiplicación	c = 3 * 6	c = 18
**	Potenciación	c = 2 ** 3	c = 8
/	División	c = 7.5 / 2	c = 3.75
//	División entera	c = 7.5 // 2	c = 3.0
%	Módulo	c = 8 % 3	c = 2

Puede que tengas dudas sobre cómo funciona el operador de módulo, y cuál es la diferencia entre división y división entera. El operador de módulo no hace otra cosa que devolvernos 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. La diferencia entre división y división entera no es otra que la que indica su nombre. En la división el resultado que se devuelve es un número real, mientras que en la división entera el resultado que se devuelve es solo la parte entera. No obstante hay que tener en cuenta que si utilizamos dos operandos enteros, Python determinará que queremos 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 operadores fuera un número real, bien indicando los decimales.

Boleanos

Como decíamos al comienzo del capítulo una variable de tipo booleano sólo puede tener dos valores: True (cierto) y False (falso). Estos valores son especialmente importantes para las expresiones condicionales y los bucles, como veremos más adelante. En realidad el tipo bool (el tipo de los booleanos) es una subclase del tipo int. Puede que esto no tenga mucho sentido para tí si no conoces los términos de la orientación a objetos, que veremos más adelante, aunque tampoco es nada importante. Estos son los distintos tipos de operadores con los que podemos trabajar con valores booleanos, los llamados operadores lógicos o condicionales.

Operador	Descripción	Ejemplo
and	¿se cumple a y b?	`r = True and False # r es False`
or	¿se cumple a o b?	`r = True or False # r es True`
not	No a	`r = not True # r es False`

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

Operador	Descripción	Ejemplo
==	¿son iguales a y b?	`r = 5 == 3 # r es False`
!=	¿son distintos a y b?	`r = 5 != 3 # r es True`
<	¿es a menor que b?	`r = 5 < 3 # r es False`
>	¿es a mayor que b?	`r = 5 > 3 # r es True`

Operadores lógicos

Estos son operadores que actúan sobre las representaciones en binario de los operandores. Por ejemplo: Si veis una operación como 3 & 2, lo que estás viendo es un and bit a bit entre los números binarios 11 y 10 (las representaciones en binario de 3 y 2). El operador and (&), del inglés "y", devuelve 1 si el primer bit operando es 1 y el segundo bit operando es 1. Se devuelve 0 en caso contrario. El resultado de aplicar and bit a bit a 11 y 10 sería entonces el número binario 10, o lo que es lo mismo, 2 en decimal (el primer dígito es 1 para ambas cifras, mientras que el segundo es 1 sólo para una de ellas). El operador or (|), del inglés "o", devuelve 1 si el primer operando es 1 o el segundo operando es 1. Para el resto de casos se devuelve 0. El operador xor u or exclusivo (^) devuelve 1 si uno de los operandos es 1 y el otro no lo es. El operador not (~), del inglés "no", sirve para negar uno a uno cada bit; es decir, si el operando es 0, cambia a 1 y si es 1, cambia a 0. Por último, los operadores de desplazamiento (<< y >>) sirven para desplazar los bits n posiciones hacia la izquierda o la derecha.

SÍMBOLO	DESCRIPCIÓN	EJEMPLO	BOOLEANO
==	IGUAL QUE	5 == 7	FALSE
!=	DISTINTO QUE	ROJO != VERDE	TRUE
<	MENOR QUE	8 < 12	TRUE
>	MAYOR QUE	12 > 7	TRUE
<=	MENOR O IGUAL QUE	16 <= 17	TRUE
>=	MAYOR O IGUAL QUE	67 >= 72	FALSE

Cadenas

Las cadenas no son más que texto encerrado entre comillas simples('cadena')o dobles ("cadena"). Dentro de las comillas se pueden añadir caracteres especiales escapándolos con \, como \n, el carácter de nueva línea, o \t, el de tabulación. Una cadena puede estar precedida por el carácter u o el carácter r, los cuales indican, respectivamente, que se trata de una cadena que utiliza codificación Unicode y 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 las expresiones regulares, como veremos en el capítulo correspondiente.

```
unicode = u"äóè"
raw = r"\n"
```

También, es posible encerrar una cadena entre triples comillas (simples o dobles). De esta forma podremos escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea que introdujimos sin tener que recurrir al carácter \n, así como las comillas sin tener que escaparlas.

```
triple = """primera linea esto se vera en otra linea"""
```

Las cadenas también admiten operadores como +, que funciona realizando una concatenación de las cadenas utilizadas como operandos y *, en la que se repite la cadena tantas veces como lo indique el número utilizado como segundo operando.

```
a = "uno"
b = "dos"
c = a + b # c es "unodos"
c = a * 3 # c es "unounouno"
```

Listas

La lista es un tipo de colección ordenada. Sería equivalente a lo que en otros lenguajes se conoce por arrays, o vectores. Las listas pueden contener cualquier tipo de dato: números, cadenas, booleanos, … y también listas. Crear una lista es tan sencillo como indicar entre corchetes, y separados por comas, los valores que queremos incluir en la lista:

```
l = [22, True, "una lista", [1, 2]]
```

Podemos acceder a cada uno de los elementos de la lista escribiendo el nombre de la lista e indicando el índice del elemento entre corchetes. Ten en cuenta sin embargo que el índice del primer elemento de la lista es 0, y no 1:

```
l = [11, False]
mi_var = l[0] # mi_var vale 11
```

Si queremos acceder a un elemento de una lista incluida dentro de otra lista tendremos que utilizar dos veces este operador, primero para indicar a qué posición de la lista exterior queremos acceder, y el segundo para seleccionar el elemento de la lista interior:

```
l = ["una lista", [1, 2]]
```

También podemos utilizar este operador para modificar un elemento de la lista si lo colocamos en la parte izquierda de una asignación:

```
l = [22, True]
l[0] = 99 # Con esto l valdrá [99, True]
```

El uso de los corchetes para acceder y modificar los elementos de una lista es común en muchos lenguajes, pero Python nos depara varias sorpresas muy agradables. Una curiosidad sobre el operador [] de Python es que podemos utilizar también números negativos. Si se utiliza un número negativo como índice, esto se traduce en que el índice empieza a contar desde el final, hacia la izquierda; es decir, con [-1] accederemos al último elemento de la lista, con [-2] al penúltimo, con [-3], al antepenúltimo, y así sucesivamente.

Tuplas

Todo lo que hemos explicado sobre las listas se aplica también a las tuplas, a excepción de la forma de definirla, para lo que se utilizan paréntesis en lugar de corchetes.

```
t = (1, 2, True, "python")
```

En realidad el constructor de la tupla es la coma, no el paréntesis, pero el intérprete muestra los paréntesis, y nosotros deberíamos utilizarlos, con claridad.

```
>>> t = 1, 2, 3
>>> type(t)
type "tuple"
```

Además hay que tener en cuenta que es necesario añadir una coma para tuplas de un solo elemento, para diferenciarlo de un elemento entre paréntesis.

```
>>> t = (1)
>>> type(t)
```

Para referirnos a elementos de una tupla, como en una lista, se usa el operador []:

```
mi_var = t[0]
# mi_var es 1 mi_var = t[0:2]
# mi_var es (1, 2)
```

Podemos utilizar el operador [] debido a que las tuplas, al igual que las listas, forman parte de un tipo de objetos llamados secuencias. Permitirme un pequeño inciso para indicaros que las cadenas de texto también son secuencias, por lo que no os extrañará que podamos hacer cosas como estas:

```
c = "hola mundo"
c[0] # h
c[5:] # mundo
c[::3] # hauo
```

Volviendo al tema de las tuplas, su diferencia con las listas estriba en que las tuplas no poseen estos mecanismos de modificación a través de funciones tan útiles de los que hablábamos al final de la anterior sección.

Diccionarios

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor. Por ejemplo, veamos un diccionario de películas y directores:

```
d = {"Love Actually ": "Richard Curtis", "Kill Bill": "Tarantino", "Amélie": "Jean-Pierre Jeunet"}
```

El primer valor se trata de la clave y el segundo del valor asociado a la clave. Como clave podemos utilizar cualquier valor inmutable: podríamos usar números, cadenas, booleanos, tuplas, ... pero no listas o diccionarios, dado que son mutables. Esto es así porque los diccionarios se implementan como tablas hash, y a la hora de introducir un nuevo par clave-valor en el diccionario se calcula el hash de la clave para después poder encontrar la entrada correspondiente rápidamente. Si se modificara el objeto clave después de haber sido introducido en el diccionario, evidentemente, su hash también cambiaría y no podría ser encontrado. La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les accede no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador [].

```
d["Love Actually "] # devuelve "Richard Curtis"
```

Al igual que en listas y tuplas también se puede utilizar este operador para reasignar valores.

```
d["Kill Bill"] = "Quentin Tarantino"
```

Sin embargo en este caso no se puede utilizar slicing, entre otras cosas porque los diccionarios no son secuencias, si no mappings (mapeados, asociaciones).

Operaciones Relacionales

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

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
```

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

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

Operador <
El operador < evalúa si el valor del lado izquierdo es menor que el valor del lado

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

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

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

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

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

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

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

Sentencias condicionales

Si un programa no fuera más que una lista de órdenes a ejecutar de forma secuencial, una por una, no tendría mucha utilidad. Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de esta condición. Aquí es donde cobran su importancia el tipo booleano y los operadores lógicos y relacionales que aprendimos en el capítulo sobre los tipos básicos de Python.

If

La forma más simple de un estamento condicional es un if (del inglés si) seguido de la condición a evaluar, dos puntos (:) y en la siguiente línea e indentado, el código a ejecutar en caso de que se cumpla dicha condición.

```
fav = "PILARES"
if fav == "PILARES":
        print "Tienes buen gusto!"
        print "Gracias"
```

Eso sí, aseguraros de que el código tal cual se ha hecho en el ejemplo, es decir, aseguraros de pulsar Tabulación antes de las dos órdenes print, dado que esta es la forma de Python de saber que nuestra intención es la de que los dos print se ejecuten sólo en el caso de que se cumpla la condición, y no la de que se imprima la primera cadena si se cumple la condición y la otra siempre, cosa que se expresaría así:

if ... else

Vamos a ver ahora un condicional algo más complicado. ¿Qué haríamos si quisiéramos que se ejecutaran unas ciertas órdenes en el caso de que la condición no se cumpliera? Sin duda podríamos añadir otro if que tuviera como condición la negación del primero:

```
if fav == "PILARES":
        print "Tienes buen gusto!"
        print "Gracias"
if fav != "PILARES":
        print "Vaya, que lástima"
```

pero el condicional tiene una segunda construcción mucho más útil:

```
if fav == "PILARES":
        print "Tienes buen gusto!"
        print "Gracias"
else:
        print "Vaya, que lástima"
```

Vemos que la segunda condición se puede sustituir con un else (del inglés: si no, en caso contrario). Si leemos el código vemos que tiene bastante sentido: "si fav es igual a mundogeek.net, imprime esto y esto, si no, imprime esto otro".

Bucles

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los bucles nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

while

El bucle while (mientras) ejecuta un fragmento de código mientras se cumpla una condición.

```
edad = 0
while edad < 18:
    edad = edad + 1
    print "Felicidades, tienes " + str(edad)
```

La variable edad comienza valiendo 0. Como la condición de que edad es menor que 18 es cierta (0 es menor que 18), se entra en el bucle. Se aumenta edad en 1 y se imprime el mensaje informando de que el usuario ha cumplido un año. Recordad que el operador + para las cadenas funciona concatenando ambas cadenas. Es necesario utilizar la función str (de string, cadena) para crear una cadena a partir del número, dado que no podemos concatenar números y cadenas, pero ya comentaremos esto y mucho más en próximos capítulos. Ahora se vuelve a evaluar la condición, y 1 sigue siendo menor que 18, por lo que se vuelve a ejecutar el código que aumenta la edad en un año e imprime la edad en la pantalla.

El bucle continuará ejecutándose hasta que edad sea igual a 18, momento en el cual la condición dejará de cumplirse y el programa continuaría ejecutando las instrucciones siguientes al bucle.

for ... in

A los que tenga experiencia previa con según que lenguajes este bucle nos va a sorprender gratamente. En Python for se utiliza como una forma genérica de iterar sobre una secuencia. Y como tal intenta facilitar su uso para este fin. Este es el aspecto de un bucle for en Python:

```
secuencia = ["uno", "dos", "tres"]
for elemento in secuencia:
    print elemento
```

Como hemos dicho los for se utilizan en Python para recorrer secuencias, por lo que vamos a utilizar un tipo secuencia, como es la lista, para nuestro ejemplo. Leamos la cabecera del bucle como si de lenguaje natural se tratara: "para cada elemento en secuencia". Y esto es exactamente lo que hace el bucle: para cada elemento que tengamos en la secuencia, ejecuta estas líneas de código. Lo que hace la cabecera del bucle es obtener el siguiente elemento de la secuencia y almacenarlo en una variable de nombre elemento. Por esta razón en la primera iteración del bucle elemento valdrá "uno", en la segunda "dos", y en la tercera "tres".

Funciones

Una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos. En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor None (nada), equivalente al null de Java. Además de ayudarnos a programar y depurar dividiendo el programa en partes las funciones también permiten reutilizar código.En Python las funciones se declaran de la siguiente forma:


```
def mi_funcion(param1, param2):
    print param1
    print param2
```

Es decir, la palabra clave def seguida del nombre de la función y entre paréntesis los argumentos separados por comas. A continuación, en otra línea, indentado y después de los dos puntos tendríamos las líneas de código que conforman el código a ejecutar por la función.

También podemos encontrarnos con una cadena de texto como primera línea del cuerpo de la función. Estas cadenas se conocen con el nombre de docstring (cadena de documentación) y sirven, como su nombre indica, a modo de documentación de la función.

```
def mi_funcion(param1, param2):"""Esta función imprime los dos valores pasados como parámetros"""
    print param1
    print param2
```

Esto es lo que imprime el operador ? de iPython o la función help del lenguaje para proporcionar una ayuda sobre el uso y utilidad de las funciones. Todos los objetos pueden tener docstrings, no solo las funciones, como veremos más adelante.

La Programación Orientada a Objetos (POO u OOP según sus siglas en inglés) es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se modelan a través de clases y objetos, y en el que nuestro programa consiste en una serie de interacciones entre estos objetos.

Clases y Objetos

Para entender este paradigma primero tenemos que comprender qué es una clase y qué es un objeto. Un objeto es una entidad que agrupa un estado y una funcionalidad relacionadas. El estado del objeto se define a través de variables llamadas atributos, mientras que la funcionalidad se modela a través de funciones a las que se les conoce con el nombre de métodos del objeto. Un ejemplo de objeto podría ser un coche, en el que tendríamos atributos como la marca, el número de puertas o el tipo de carburante y métodos como arrancar y parar. O bien cualquier otra combinación de atributos y métodos según lo que fuera relevante para nuestro programa. Una clase, por otro lado, no es más que una plantilla genérica a partir de la cuál instanciar los objetos; plantilla que es la que define qué atributos y métodos tendrán los objetos de esa clase. Volviendo a nuestro ejemplo: en el mundo real existe un conjunto de objetos a los que llamamos coches y que tienen un conjunto de atributos comunes y un comportamiento común, esto es a lo que llamamos clase. Sin embargo, mi coche no es igual que el coche de mi vecino, y aunque pertenecen a la misma clase de objetos, son objetos distintos. En Python las clases se definen mediante la palabra clave class seguida del nombre de la clase, dos puntos (:) y a continuación, indentado, el cuerpo de la clase. Como en el caso de las funciones, si la primera línea del cuerpo se trata de una cadena de texto, esta será la cadena de documentación de la clase o docstring.

```
class Coche:
    """Abstraccion de los objetos coche."""
    def __init__(self, gasolina):
        self.gasolina = gasolina
        print "Tenemos", gasolina, "litros"

    def arrancar(self):
        if self.gasolina > 0:
            print "Arranca"
        else:
            print "No arranca"

    def conducir(self):
        if self.gasolina > 0:
            self.gasolina -= 1
            print "Quedan", self.gasolina, "litros"
        else:
            print "No se mueve"
```

Lo primero que llama la atención en el ejemplo anterior es el nombre tan curioso que tiene el método __init__. Este nombre es una convención y no un capricho. El método __init__, con una doble barra baja al principio y final del nombre, se ejecuta justo después de crear un nuevo objeto a partir de la clase, proceso que se conoce con el nombre de instanciación. El método __init__ sirve, como sugiere su nombre, para realizar cualquier proceso de inicialización que sea necesario. Como vemos el primer parámetro de __init__ y del resto de métodos de la clase es siempre self. Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local mi_var de un atributo del objeto self.mi_var.

```
mi_coche = Coche(3)
```

Si volvemos al método __init__ de nuestra clase Coche veremos cómo se utiliza self para asignar al atributo gasolina del objeto (self.gasolina) el valor que el programador especificó para el parámetro gasolina. El parámetro gasolina se destruye al final de la función, mientras que el atributo gasolina se conserva (y puede ser accedido) mientras el objeto viva. Para crear un objeto se escribiría el nombre de la clase seguido de cualquier parámetro que sea necesario entre paréntesis. Estos parámetros son los que se pasarán al método __init__, que como decíamos es el método que se llama al instanciar la clase.

```
>>> print mi_coche.gasolina
3
>>> mi_coche.arrancar()
Arranca
>>> mi_coche.conducir()
Quedan 2 litros
>>> mi_coche.conducir()
Quedan 1 litros
>>> mi_coche.conducir()
Quedan 0 litros
>>> mi_coche.conducir()
```

Entonces, cómo es posible que a la hora de crear nuestro primer objeto pasemos un solo parámetro a __init__, el número 3, cuando la definición de la función indica claramente que precisa de dos parámetros (self y gasolina). Esto es así porque Python pasa el primer argumento (la referencia al objeto que se crea) automáticamente. Ahora que ya hemos creado nuestro objeto, podemos acceder a sus atributos y métodos mediante la sintaxis objeto.atributo y objeto. método():

Herencias

Hay tres conceptos que son básicos para cualquier lenguaje de programación orientado a objetos: el encapsulamiento, la herencia y el polimorfismo. En un lenguaje orientado a objetos cuando hacemos que una clase (subclase) herede de otra clase (superclase) estamos haciendo que la subclase contenga todos los atributos y métodos que tenía la superclase. No obstante al acto de heredar de una clase también se le llama a menudo "extender una clase". Supongamos que queremos modelar los instrumentos musicales de una banda, tendremos entonces una clase Guitarra, una clase Batería, una clase Bajo, etc. Cada una de estas clases tendrá una serie de atributos y métodos, pero ocurre que, por el mero hecho de ser instrumentos musicales, estas clases compartirán muchos de sus atributos y métodos; un ejemplo sería el método tocar(). Es más sencillo crear un tipo de objeto Instrumento con las atributos y métodos comunes e indicar al programa que Guitarra, Batería y Bajo son tipos de instrumentos, haciendo que hereden de Instrumento. Para indicar que una clase hereda de otra se coloca el nombre de la clase de la que se hereda entre paréntesis después del nombre de la clase:

```
class Instrumento:
 def __init__(self, precio):
 self.precio = precio
 def tocar(self):
 print "Estamos tocando musica"
 def romper(self):
 print "Eso lo pagas tu"
 print "Son", self.precio, "$$$"
class Bateria(Instrumento):
 pass
class Guitarra(Instrumento):
 pass
```

Cómo Batería y Guitarra heredan de Instrumento, ambos tienen un método tocar() y un método romper(), y se inicializan pasando un parámetro precio. Pero, ¿qué ocurriría si quisiéramos especificar un nuevo parámetro tipo_cuerda a la hora de crear un objeto Guitarra? Bastaría con escribir un nuevo método __init__ para la clase Guitarra que se ejecutaría en lugar del __init__ de Instrumento. Esto es lo que se conoce como sobreescribir métodos. Ahora bien, puede ocurrir en algunos casos que necesitemos sobreescribir un método de la clase padre, pero que en ese método queramos ejecutar el método de la clase padre porque nuestro nuevo método no necesite más que ejecutar un par de nuevas instrucciones extra. En ese caso usamos la sintaxis SuperClase.metodo(self, args) para llamar al método de igual nombre de la clase padre. Por ejemplo, para llamar al método __init__ de Instrumento desde Guitarra usamos Instrumento.__init__(self, precio) Observa que en este caso si es necesario especificar el parámetro self.

Herencias Múltiples

En Python, a diferencia de otros lenguajes como Java o C#, se permite la herencia múltiple, es decir, una clase puede heredar de varias clases a la vez. Por ejemplo, podríamos tener una clase Cocodrilo que heredara de la clase Terrestre, con métodos como caminar() y atributos como velocidad_caminar y de la clase Acuatico, con métodos como nadar() y atributos como velocidad_nadar. Basta con enumerar las clases de las que se hereda separándolas por comas:

```
class Cocodrilo(Terrestre, Acuatico):
    pass
```

En el caso de que alguna de las clases padre tuvieran métodos con el mismo nombre y número de parámetros las clases sobrescribirá la implementación de los métodos de las clases más a su derecha en la definición. En el siguiente ejemplo, como Terrestre se encuentra más a la izquierda, sería la definición de desplazar de esta clase la que prevalecerá, y por lo tanto si llamamos al método desplazar de un objeto de tipo Cocodrilo lo que se imprimiría sería "El animal anda".

```
class Terrestre:
    def desplazar(self):
        print "El animal anda"

class Acuatico:
    def desplazar(self):
        print "El animal nada"

class Cocodrilo(Terrestre, Acuatico):
    pass

c = Cocodrilo()
c.desplazar()
```

Polimorfismo

El polimorfismo es uno de los pilares básicos en la programación orientada a objetos, por lo que para entenderlo es importante tener las bases de la POO y la herencia bien asentada. El término polimorfismo tiene origen en las palabras poly (muchos) y morfo (formas), y aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas. ¿Pero qué significa esto? Pues bien, significa que objetos de diferentes clases pueden ser accedidos utilizando el mismo interfaz, mostrando un comportamiento distinto (tomando diferentes formas) según cómo sean accedidos. En lenguajes de programación como Python, que tiene tipado dinámico, el polimorfismo va muy relacionado con el duck typing. Sin embargo, para entender bien este concepto, es conveniente explicarlo desde el punto de vista de un lenguaje de programación con tipado estático como Java. Vamos a por ello.
Polimorfismo en Java Vamos a comenzar definiendo una clase Animal.

```java
// Código Java
class Animal{
    public Animal() {}
}
```

Y dos clases Perro y Gato que heredan de la anterior.

```java
// Código Java
class Perro extends Animal {
    public Perro() {}
}

class Gato extends Animal {
    public Gato() {}
}
```

El polimorfismo es precisamente lo que nos permite hacer lo siguiente:

```java
// Código Java
Animal a = new Perro();
```

Recuerda que Java es un lenguaje con tipado estático, lo que significa que el tipo tiene que ser definido al crear la variable. Sin embargo, estamos asignando a una variable Animal un objeto de la clase Perro. ¿Cómo es esto posible? Pues ahí lo tienes, el polimorfismo es lo que nos permite usar ambas clases de forma indistinta, ya que soportan el mismo interfaz (no confundir con el interface de Java).El siguiente código es también correcto. Tenemos un array de Animal donde cada elemento toma la forma de Perro o de Gato.

```java
// Código Java
Animal[] animales = new Animal[10];
animales[0] = new Perro();
animales[1] = new Gato();
```

Sin embargo, no es posible realizar lo siguiente, ya que OtraClase no comparte interfaz con Animal. Tendremos un error error: incompatible types.

```java
// Código Java
class OtraClase {
    public OtraClase() {}
}
Animal a = new OtraClase();
animales[0] = new OtraClase();
```

Polimorfismo en Python

El término polimorfismo visto desde el punto de vista de Python es complicado de explicar sin hablar del duck typing, por lo que te recomendamos la lectura. Al ser un lenguaje con tipado dinámico y permitir duck typing, en Python no es necesario que los objetos compartan un interfaz, simplemente basta con que tengan los métodos que se quieren llamar.

Podemos recrear el ejemplo de Java de la siguiente manera. Supongamos que tenemos un clase Animal con un método hablar().

```
class Animal:
    def hablar(self):
        pass
```

Por otro lado tenemos otras dos clases, Perro, Gato que heredan de la anterior. Además, implementan el método hablar() de una forma distinta.

```
class Perro(Animal):
    def hablar(self):
        print("Guau!")

class Gato(Animal):
    def hablar(self):
        print("Miau!")
```

A continuación creamos un objeto de cada clase y llamamos al método hablar(). Podemos observar que cada animal se comporta de manera distinta al usar hablar().

```
for animal in Perro(), Gato():
    animal.hablar()

# Guau!
# Miau!
```

Encapsulación

La encapsulación se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase. Esto se consigue en otros lenguajes de programación como Java utilizando modificadores de acceso que definen si cualquiera puede acceder a esa función o variable (public) o si está restringido el acceso a la propia clase (private). En Python no existen los modificadores de acceso, y lo que se suele hacer es que el acceso a una variable o función viene determinado por su nombre: si el nombre comienza con dos guiones bajos (y no termina también con dos guiones bajos) se trata de una variable o función privada, en caso contrario es pública.

Los métodos cuyo nombre comienza y termina con dos guiones bajos son métodos especiales que Python llama automáticamente bajo ciertas circunstancias.

En el siguiente ejemplo sólo se imprimirá la cadena correspondiente al método público(), mientras que al intentar llamar al método __privado() Python lanzará una excepción quejándose de que no existe (evidentemente existe, pero no lo podemos ver porque es privado).

```
class Ejemplo:
    def publico(self):
        print "Publico"

    def __privado(self):
        print "Privado"

ej = Ejemplo()
ej.publico()
ej.__privado()
```

Este mecanismo se basa en que los nombres que comienzan con un doble guión bajo se renombran para incluir el nombre de la clase (característica que se conoce con el nombre de name mangling). Esto implica que el método o atributo no es realmente privado, y podemos acceder a él mediante una pequeña trampa:

```
ej._Ejemplo__privado()
```

En ocasiones también puede suceder que queramos permitir el acceso a algún atributo de nuestro objeto, pero que este se produzca de forma controlada. Para esto podemos escribir métodos cuyo único cometido sea este, métodos que normalmente, por convención, tienen nombres como getVariable y setVariable; de ahí que se conozcan también con el nombre de getters y setters.

```
class Fecha():
    def __init__(self):
        self.__dia = 1

    def getDia(self):
        return self.__dia

    def setDia(self, dia):
        if dia > 0 and dia < 31:
            self.__dia = dia
        else:
            print "Error"

mi_fecha = Fecha()
mi_fecha.setDia(33)
```

Esto se podría simplificar mediante propiedades, que abstraen al usuario del hecho de que se está utilizando métodos entre bambalinas para obtener y modificar los valores del atributo:

```
class Fecha(object):
    def __init__(self):
        self.__dia = 1

    def getDia(self):
        return self.__dia

    def setDia(self, dia):
        if dia > 0 and dia < 31:
            self.__dia = dia
        else:
            print "Error"

    dia = property(getDia, setDia)

mi_fecha = Fecha()
mi_fecha.dia = 33
```

Métodos de los Objetos.

S.count(sub[, start[, end]])	Devuelve el número de veces que se encuentra sub en la cadena. Los parámetros opcionales start y end definen una subcadena en la que buscar.
S.find(sub[, start[, end]])	Devuelve la posición en la que se encontró por primera vez sub en la cadena o -1 si no se encontró.
S.join(sequence)	Devuelve una cadena resultante de concatenar las cadenas de la secuencia se que separadas por la cadena sobre la que se llama el método.
S.partition(sep)	Busca el separador sep en la cadena y devuelve una tupla con la subcadena hasta dicho separador, el separador en si, y la subcadena del separador hasta el final de la cadena. Si no se encuentra el separador, la tupla contendrá la cadena en si y dos cadenas vacías.
S.replace(old, new[, count]	Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena old por la cadena new. Si se especifica el parámetro count, este indica el número máximo de ocurrencias a reemplazar.
S.split([sep [,maxsplit]])	Devuelve una lista conteniendo las subcadenas en las que se divide nuestra cadena al dividirlas por el delimitador sep. En el caso de que no se especifique sep, se usan espacios. Si se especifica maxsplit, este indica el número máximo de particiones a realizar.

Métodos de los Objetos.

Listas

L.append(object)	Añade un objeto al final de la lista.
L.count(value)	Devuelve el número de veces que se encontró value en la lista.
L.extend(iterable)	Añade los elementos del iterable a la lista.
L.index(value[, start[, stop]])	Devuelve la posición en la que se encontró la primera ocurrencia de value. Si se especifican, start y stop definen las posiciones de inicio y fin de una sublista en la que buscar.
L.insert(index, object)	Inserta el objeto object en la posición index.
L.pop([index])	Devuelve el valor en la posición index y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.
L.remove(value)	Eliminar la primera ocurrencia de value en la lista.
L.reverse()	Invierte la lista. Esta función trabaja sobre la propia lista desde la que se invoca el método, no sobre una copia.
L.sort(cmp=None, key=None, reverse=False)	Ordena la lista. Si se especifica cmp, este debe ser una función que tome como parámetro dos valores x e y de la lista y devuelva -1 si x es menor que y, 0 si son iguales y 1 si x es mayor que y. El parámetro reverse es un booleano que indica si se debe ordenar la lista de forma inversa, lo que sería equivalente a llamar primero a L.sort() y después a L.reverse().

Métodos de los Objetos.

D.get(k[, d])	Busca el valor de la clave k en el diccionario. Es equivalente a utilizar D[k] pero al utilizar este método podemos indicar un valor a devolver por defecto si no se encuentra la clave, mientras que con la sintaxis D[k], de no existir la clave se lanzaría una excepción.
D.has_key(k)	Comprueba si el diccionario tiene la clave k. Es equivalente a la sintaxis k in D.
D.items()	Devuelve una lista de tuplas con pares clave-valor. D.keys() Devuelve una lista de las claves del diccionario.
D.pop(k[, d])	Borra la clave k del diccionario y devuelve su valor. Si no se encuentra dicha clave se devuelve d si se especificó el parámetro o bien se lanza una excepción.D.values() Devuelve una lista de los valores del diccionario.

La programación funcional es un paradigma en el que la programación se basa casi en su totalidad en funciones, entendiendo el concepto de función según su definición matemática, y no como los simples subprogramas de los lenguajes imperativos que hemos visto hasta ahora. En los lenguajes funcionales puros un programa consiste exclusivamente en la aplicación de distintas funciones a un valor de entrada para obtener un valor de salida. Python, sin ser un lenguaje puramente funcional incluye varias características tomadas de los lenguajes funcionales como son las funciones de orden superior o las funciones lambda (funciones anónimas).

Función de orden superior

El concepto de funciones de orden superior se refiere al uso de funciones como si de un valor cualquiera se tratara, posibilitando el pasar funciones como parámetros de otras funciones o devolver funciones como valor de retorno. Es posible porque, como hemos insistido ya en varias ocasiones, en Python todo son objetos. Y las funciones no son una excepción. Veamos un pequeño ejemplo:

```
1   def saludar(lang):
2    def saludar_es():
3   Python para todos
4   58
5    print “Hola”
6    def saludar_en():
7    print “Hi”
8    def saludar_fr():
9    print “Salut”
10   lang_func = {“es”: saludar_es,
11   “en”: saludar_en,
12   “fr”: saludar_fr}
13   return lang_func[lang]
14  f = saludar(“es”)
15  f()
```

Como podemos observar lo primero que hacemos en nuestro pequeño programa es llamar a la función saludar con un parámetro “es”. En la función saludar se definen varias funciones: saludar_es, saludar_en y saludar_fr y a continuación se crea un diccionario que tiene como claves cadenas de texto que identifican a cada lenguaje, y como valores las funciones. El valor de retorno de la función es una de estas funciones.

La función para devolver viene determinada por el valor del parámetro lang que se pasó como argumento de saludar. Como el valor de retorno de saludar es una función, como hemos visto, esto quiere decir que f es una variable que contiene una función. Podemos entonces llamar a la función a la que se refiere f de la forma en que llamaríamos a cualquier otra función, añadiendo unos paréntesis y, de forma opcional, una serie de parámetros entre los paréntesis. Esto se podría acortar, ya que no es necesario almacenar la función que nos pasan como valor de retorno en una variable para poder llamarla:

```
>>> saludar("en")()
Hi
>>> saludar("fr")()
Salut
```

MAP

Esta función trabaja de una forma muy similar a filter(), 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 iterable de tipo map:

Filter

Tal como su nombre indica filter significa filtrar, y es una de mis funciones favoritas, 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. Por ejemplo, supongamos que tenemos una lista de varios números y queremos filtrarla, quedándonos únicamente con los múltiplos de 5:

```
1  def multiple(numero):    # Primero declaramos una función condicional
2      if numero % 5 == 0:  # Comprobamos si un numero es múltiple de cinco
3          return True      # Sólo devolvemos True si lo es
4
5  numeros = [2, 5, 10, 23, 50, 33]
6
7  filter(multiple, numeros)
8  <filter at 0x257ac84abe0>
```

Si ejecutamos el filtro obtenemos un objeto de tipo filtro, pero podemos transformarlo en una lista fácilmente haciendo un cast (conversión):

```
list( filter(multiple, numeros) )
```

```
[5, 10, 50]
```

Por tanto cuando utilizamos la función filter() tenemos que enviar una función condicional, pero como recordaréis, no es necesario definirla, podemos utilizar una función anónima lambda:

```
list( filter(lambda numero: numero%5 == 0, numeros) )
```

```
[5, 10, 50]
```

Así, en una sola línea hemos definido y ejecutado el filtro utilizando una función condicional anónima y una lista de números.

Reduce

Reduce es una función incorporada de Python 2, que toma como argumento un conjunto de valores (una lista, una tupla, o cualquier objeto iterable) y lo "reduce" a un único valor. Cómo se obtiene ese único valor a partir de la colección pasada como argumento dependerá de la función aplicada. Por ejemplo, el siguiente código reduce la lista [1, 2, 3, 4] al número 10 aplicando la función add(a, b), que retorna la suma de sus argumentos.

```
def add(a, b):
    return a + b

print(reduce(add, [1, 2, 3, 4]))  # 10
```

La función pasada como primer argumento debe tener dos parámetros. reduce() se encargará de llamarla de forma acumulativa (es decir, preservando el resultado de llamadas anteriores) de izquierda a derecha. De modo que el código anterior es similar a:

```
print(add(add(add(1, 2), 3), 4))
```

Funciones lamba

En Python, una función Lambda se refiere a una pequeña función anónima. Las llamamos “funciones anónimas” porque técnicamente carecen de nombre. Al contrario que una función normal, no la definimos con la palabra clave estándar def que utilizamos en Python. En su lugar, las funciones Lambda se definen como una línea que ejecuta una sola expresión. Este tipo de funciones pueden tomar cualquier número de argumentos, pero solo pueden tener una expresión.
Sintaxis básica
Todas las funciones Lambda en Python tienen exactamente la misma sintaxis

```
#Escribo p1 y p2 como parámetros 1 y 2 de la función.

lambda p1, p2: expresión
```

Como mejor te lo puedo explicar es enseñándote un ejemplo básico, vamos a ver una función normal y un ejemplo de Lambda:

```
#Aquí tenemos una función creada para sumar.
def suma(x,y):
    return(x + y)

#Aquí tenemos una función Lambda que también suma.
lambda x,y : x + y

#Para poder utilizarla necesitamos guardarla en una variable.
suma_dos = lambda x,y : x + y
```

Al igual que ocurre en las list, lo que hemos hecho es escribir el código en una sola línea y limpiar la sintaxis innecesaria. En lugar de usar def para definir nuestra función, hemos utilizado la palabra clave lambda; a continuación escribimos x, y como argumentos de la función, y x + y como expresión. Además, se omite la palabra clave return, condensando aún más la sintaxis. Por último, y aunque la definición es anónima, la almacenamos en la variable suma_dos para poder llamarla desde cualquier parte del código, de no ser así tan solo podríamos hacer uso de ella en la línea donde la definamos.

Comprensión de listas

La comprensión de listas, del inglés list comprehensions, es una funcionalidad que nos permite crear listas avanzadas en una misma línea de código. Esto se ve mucho mejor en la práctica, así que a lo largo de esta lección vamos a trabajar distintos ejemplos.

Crear una lista con las letras de una palabra:

```
# Método tradicional
lista = []
for letra in 'casa':
    lista.append(letra)
print(lista)
```

```
['c', 'a', 's', 'a']
```

Como vemos, gracias a la comprensión de listas podemos indicar directamente cada elemento que va a formar la lista, en este caso la letra, a la vez que definimos el for:

```
# Con comprensión de listas
lista = [letra for letra in 'casa']
print(lista)
```

```
['c', 'a', 's', 'a']
```

Crear una lista con las potencias de 2 de los primeros 10 números:

```
# Método tradicional
lista = []
for numero in range(0,11):
    lista.append(numero**2)
print(lista)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

De este ejemplo podemos aprender que es posible modificar al vuelo los elementos que van a formar la lista.

```
# Con comprensión de listas
lista = [numero**2 for numero in  range(0,11)]
print(lista)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Generadores

Un generador es una función que produce una secuencia de resultados en lugar de un único valor. Es decir, cada vez que llamemos a la función nos darán un nuevo resultado. Para construir generadores sólo tenemos que usar la orden yield. Esta orden devolverá un valor igual que hace return pero, además, pasará la ejecución de la función hasta la próxima vez que le pidamos un valor.

```
[numero for numero in range(0,11) if numero % 2 == 0 ]
```

```
[0, 2, 4, 6, 8, 10]
```

La verdad es que range es una especie de función generadora. Por regla general las funciones devuelven un valor con return, pero la peculiaridad de los generadores es que van cediendo valores sobre la marcha, en tiempo de ejecución. La función generadora range(0,11), empieza cediendo el 0, luego se procesa el for comprobando si es par y lo añade a la lista, en la siguiente iteración se cede el 1, se procesa el for se comprueba si es par, en la siguiente se cede el 2, etc. Con esto se logra ocupar el mínimo de espacio en la memoria y podemos generar listas de millones de elementos sin necesidad de almacenarlos previamente

Veamos a ver cómo crear una función generadora de pares:

```
def pares(n):
    for numero in range(n+1):
        if numero % 2 == 0:
            yield numero

pares(10)
```

Como vemos, en lugar de utilizar el return, la función generadora utiliza el yield, que significa ceder. Tomando un número busca todos los pares desde o hasta el número+1 sirviéndose de un range(). Sin embargo, fijaros que al imprimir el resultado, éste nos devuelve un objeto de tipo generador. De la misma forma que recorremos un range() podemos utilizar el bucle for para recorrer todos los elementos que devuelve el generador.

Clases Decoradoras

Los decoradores son un patrón de diseño de software que alteran dinámicamente y agregan funcionalidades adicionales a los métodos, funciones o clases de Python sin tener que usar subclases o cambiar el código fuente decorada. Estos generalmente son herramientas muy útiles para el desarrollador ya que son muy fáciles de implementar, son legibles, reducen código, entre otras ventajas. En Python hay múltiples formas de crear decoradores y en este tutorial vamos a abarcar todas esas maneras mediante ejemplos prácticos y explicaciones detalladas.

```
1   def decorator(func):
2     print("Decorator")
3     return func
4
5   @decorator
6   def Hello():
7     print("Hello World")
8
9   Hello()
10  # [Output]:
11  # Decorator
12  # Hello World
```

La notación del decorador es mediante el símbolo @ seguido del nombre de la función que cumple el papel de decorador, esto equivale a hacer esto:

```
Hello = decorator(Hello)
```

Es importante tener esto en cuenta para comprender cómo funcionan los decoradores. Esta sintaxis deja bien en claro el porqué los decoradores toman una función como argumento y luego retornan también una función. ¿Qué pasa si no retorna nada?

```
1   def disabled(func):
2    """
3    Esta función no retorna nada, por tanto, no se ejecuta la función decorada
4    """
5    pass
6
7   @disabled
8   def foobar():
9    print("Esta función no se ejecuta")
0
1   foobar()
2   # TypeError: 'NoneType' object is not callable
```

Como lo observamos con el ejemplo anterior si el decorador no retorna la función, esta se ejecutará y lanzará un Error NoneType

Excepciones

Los errores de ejecución son llamados comúnmente excepciones y por eso de ahora en más utilizaremos ese nombre. Durante la ejecución de un programa, si dentro de una función surge una excepción y la función no la maneja, la excepción se propaga hacia la función que la invocó, si esta otra tampoco la maneja, la excepción continúa propagándose hasta llegar a la función inicial del programa y si esta tampoco la maneja se interrumpe la ejecución del programa. Veamos entonces cómo manejar excepciones. Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa. En el caso de Python, el manejo de excepciones se hace mediante los bloques que utilizan las sentencias try, except y finally.

```
>>> dividendo = 5
>>> divisor = 0
>>> dividendo / divisor
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

En este caso, se levantó la excepción ZeroDivisionError cuando se quiso hacer la división. Para evitar que se levante la excepción y se detenga la ejecución del programa, se utiliza el bloque try-except.

```
>>> try:
...     cociente = dividendo / divisor
... except:
...     print "No se permite la división por cero"
...
```

No se permite la división por cero Dado que dentro de un mismo bloque try pueden producirse excepciones de distinto tipo, es posible utilizar varios bloques except, cada uno para capturar un tipo distinto de excepción.

Esto se hace especificando a continuación de la sentencia except el nombre de la excepción que se pretende capturar. Un mismo bloque except puede atrapar varios tipos de excepciones, lo cual se hace especificando los nombres de la excepciones separados por comas a continuación de la palabra except. Es importante destacar que si bien luego de un bloque try puede haber varios bloques except, se ejecutará, a lo sumo, uno de ellos.

Entrada Estándar rawInput

Entrada de datos por teclado en Python Python proporciona dos funciones integradas para leer una línea de texto de la entrada estándar por teclado. Estas funciones son: raw_input() input() La función raw_input El raw_input ([indicacion]) función lee una línea de la entrada estándar y lo devuelve como una cadena (quitando el salto de línea final). Sintaxis : str= raw_input('interaccion ')

```
>>> str = raw_input("Ingrese un host a atacar: ")
Ingrese un host a atacar: 192.168.1.3
>>> print "atacando a " + str
atacando a 192.168.1.3
>>>
```

Cabe señalar que la función raw_input() recibe y procesa de forma literal lo ingresado por teclado, por ejemplo si ingresamos 1+2 +4 el intérprete lo maneja como texto y no lo procesa:

Ejemplo:

```
>>> str = raw_input("Ingrese algo por teclado: ")
Ingrese algo por teclado: 1+2+4
>>> print str
1+2+4
>>>
```

La Función input

La función input([indicacion]) es equivalente a raw_input, excepto que devuelve el resultado evaluado en su caso.

```
>>> str = input("Ingrese algo por teclado: ")
Ingrese algo por teclado: 1+2+4
>>> print str
7
>>>
```

Al recibir números, el intérprete los proceso ejecutándose como una operación matemática. Otro ejemplo tratando de mostrar la diferencia que existe entre la función input y raw_input es ingresando por ejemplo una dirección ip:

```
>>> str = input("Ingrese algo por teclado: ")
Ingrese algo por teclado: 192.168.1.1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1
    192.168.1.1
            ^
SyntaxError: invalid syntax
>>>
```

El intérprete al intentar procesar una dirección ip, se da cuanta que son números y los trabaja como tal, por lo que la función input, no sirve mucho para hacer nuestros programas, recomiendo utilizar casi siempre raw_input.

Módulos

Para facilitar el mantenimiento y la lectura los programas demasiado largos pueden dividirse en módulos, agrupando elementos relacionados. Los módulos son entidades que permiten una organización y división lógica de nuestro código. Los ficheros son su contrapartida física: cada archivo Python almacenado en disco equivale a un módulo. Vamos a crear nuestro primer módulo entonces creando un pequeño archivo modulo.py con el siguiente contenido:

```
def mi_funcion():
    print "una funcion"

class MiClase:
    def __init__(self):
        print "una clase"

print "un modulo"
```

Si quisiéramos utilizar la funcionalidad definida en este módulo en nuestro programa tendríamos que importarlo. Para importar un módulo se utiliza la palabra clave import seguida del nombre del módulo, que consiste en el nombre del archivo menos la extensión. Como ejemplo, creemos un archivo programa.py en el mismo directorio en el que guardamos el archivo del módulo (esto es importante, porque si no se encuentra en el mismo directorio Python no podrá encontrarlo), con el siguiente contenido:

```
import modulo


modulo.mi_funcion()
```

El import no solo hace que tengamos disponible todo lo definido dentro del módulo, sino que también ejecuta el código del módulo. Por esta razón nuestro programa, además de imprimir el texto "una función" al llamar a mi_funcion, también imprimiría el texto "un módulo", debido al print del módulo importado. No se imprimiría, no obstante, el texto "una clase", ya que lo que se hizo en el módulo fue tan solo definir de la clase, no instanciarla. La clausula import también permite importar varios módulos en la misma línea. En el siguiente ejemplo podemos ver cómo se importa con una sola clausula import los módulos de la distribución por defecto de Python os, que engloba funcionalidad relativa al sistema operativo; sys, con funcionalidad relacionada con el propio intérprete de Python y time, en el que se almacenan funciones para manipular fechas yhoras.