UD1: Introducción a Python

Índice

Introducción

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**.

- https://www.youtube.com/watch?v=o54Nxtp47NA

Lenguaje Interpretado o de script

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).

La ventaja de los lenguajes compilados es que su ejecución es más rápida. Sin embargo los lenguajes interpretados son más flexibles y más portables.

Python tiene, no obstante, muchas de las características de los lenguajes compilados, por lo que se podría decir que es semi interpretado. En Python, como en Java y muchos otros lenguajes, el código fuente se traduce a un pseudo código máquina intermedio llamado bytecode la primera vez que se ejecuta, generando archivos .pyc o .pyo (bytecode optimizado), que son los que se ejecutarán en sucesivas ocasiones.

Tipado dinámico

La característica de tipado dinámico se refiere a que **no es necesario declarar el tipo de dato que va a contener una determinada variable**, sino que **su tipo se determinará en tiempo de ejecución** según el tipo del valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.

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.

¿Por qué Python?

Python es un lenguaje que todo el mundo debería conocer. Su sintaxis simple, clara y sencilla; el tipado dinámico, el gestor de memoria, la gran cantidad de librerías disponibles y la potencia del lenguaje, entre otros, hacen que desarrollar una aplicación en Python sea sencillo, muy rápido y, lo que es más importante, divertido.

La sintaxis de Python es tan sencilla y cercana al lenguaje natural que los programas elaborados en Python parecen pseudocódigo. Por este motivo se trata además de uno de los mejores lenguajes para comenzar a programar.

Python no es adecuado sin embargo para la programación de bajo nivel o para aplicaciones en las que el rendimiento sea crítico.

Algunos casos de éxito en el uso de Python son Google, Yahoo, la NASA, Industrias Light & Magic, y todas las distribuciones Linux, en las que Python cada vez representa un tanto por ciento mayor de los programas disponibles.

Tipos de datos

Tipos de datos básicos

Python dispone de 3 tipos básicos de datos:

- **Número**: Python soporta la representación de números Enteros (int y long), Reales (float y double) y Complejos.
- **Cadenas**:
- **Booleano**: Los datos booleanos pueden tener dos valores: True y False.

Números

- **Enteros**:
  - Int: Depende de la plataforma del equipo en el que se ejecute el programa ocupa 32 o 64 bits en memoria.
    - *# type(entero) devolvería int*
    - *entero = 23*
  - Long: Ocupa todo el espacio que necesite o esté disponible en memoria. Solo utilizar cuando sea necesario. Si ponemos una L al final de un número entero indicamos que dicho número es de tipo long
    - *# type(entero) devolvería long*
    - *entero = 23L*

Números

- **Reales**: El tipo float ocupa 64 bits en memoria y el tipo double 128.

- Los números reales se pueden representar de estas dos maneras:
  - real = 1.235
  - real = 1235e-3 (notación científica $1235x10^{-3}$ = 1235x0,001 = 1.235)

- **Complejos**: Son aquellos que tienen parte imaginaria. (Es poco probable que los utilicemos)
  - complejo = 3.2 + 4.2j

Números - Operadores

Los operadores numéricos en Python son lo siguientes:

Descripción	Operador	Ejemplo	
Suma	+	r = 3+6	#r es 9
Resta / Negación	-	r = 9-5 r = -5	#r es 4 #r es -5
Multiplicación	*	r = 3 * 5	#r es 15
Exponente	**	r = 5 ** 3	#r es 125
División	/	r = 7 / 2	#r es 3.5
División entera	//	r = 7 // 2	#r es 3
Módulo	%	r = 7 % 2	#r es 1

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 fabulación

Una cadena puede estar precedida por el carácter **u** (codificación Unicode) o el carácter **r** (cadena raw). 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.

unicode = u”äóè”
raw = r”\n”

Cadenas

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 **+** (concatenación) y ***** (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"
```

Booleanos

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. Las operaciones relacionadas con los tipos de datos booleanos son **and**, **or** y **not**.

Además, el resultado de algunas expresiones relacionales dan como resultado un valor booleano.

Descripción	Operador	Ejemplo	
¿son iguales?	==	r = 5 == 3	#r es False
¿son diferentes?	!=	r = 5 != 3	#r es True
¿es menor?	<	r = 5 < 3	#r es False
¿es mayor?	>	r = 5 > 3	#r es True
¿es menor o igual?	<=	r = 5 <= 3	#r es False
¿es mayor o igual	>=	r = 5 >= 3	#r es True

Colecciones

Colecciones

Python tiene varias tipos de colecciones de datos:

- **Listas**: La lista es un tipo de colección ordenada. Sería equivalente a lo que en otros lenguajes se conoce por arrays, o vectores.
- **Tuplas**: Son iguales que las listas pero con una gran diferencia, son inmutables, es decir, sus valores no se pueden modificar una vez creada; y tienen un tamaño fijo.
- **Diccionarios**: Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor.
- **Conjuntos**: Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas.

Listas

La lista es un tipo de colección ordenada (Equivalente a los arrays).

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]]

Listas

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. (El índice del primer elemento de la lista es 0):

```
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]]
mi_var = l[1][0]        # mi_var vale 1
```

Listas

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]
```

Con 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] accederíamos al último elemento de la lista, con [-2] al penúltimo, con [-3] , al antepenúltimo, y así sucesivamente.

Listas

Otra característica es el **slicing o particionado**, y que permite seleccionar porciones de la lista. Si en lugar de un número escribimos dos números inicio y fin separados por dos puntos (**inicio:fin**) Python interpretará que queremos una lista que vaya desde la posición inicio a la posición fin, sin incluir este último. Si escribimos tres números (**inicio:fin:salto**) en lugar de dos, el tercero se utiliza para determinar cada cuantas posiciones añadir un elemento a la lista.

```
l = [99, True, “una lista”, [1, 2]]
mi_var = l[0:2] # mi_var vale [99, True]
mi_var = l[0:4:2] # mi_var vale [99, “una lista”]
```

Listas

Los números negativos también se pueden utilizar en un slicing, con el mismo comportamiento que se comentó anteriormente.

Hay que mencionar así mismo que no es necesario indicar el principio y el final del slicing, sino que, si estos se omiten, se usarán por defecto las posiciones de inicio y fin de la lista, respectivamente:

```
l = [99, True, “una lista”]
mi_var = l[1:]      # mi_var vale [True, “una lista”]
mi_var = l[:2]      # mi_var vale [99, True]
mi_var = l[:]       # mi_var vale [99, True, “una lista”]
mi_var = l[::2]     # mi_var vale [99, “una lista”]
```

Listas

Las cadenas de texto son secuencias (y por lo tanto comparten características con las listas), por lo que no os extrañará que podamos hacer cosas como estas:

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

Tuplas

La principal característica (y diferencia con las listas) e que son inmutables, es decir, sus valores no se pueden modificar una vez creada; y tienen un tamaño fijo.

A cambio de estas limitaciones las tuplas son más “ligeras” que las listas, por lo que si el uso que le vamos a dar a una colección es muy básico, puedes utilizar tuplas en lugar de listas y ahorrar memoria.

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”)

Tuplas

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

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

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)
```

Diccionarios

Los diccionarios, también llamados matrices asociativas, deben su nombre a que son colecciones que relacionan una clave y un valor.

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.

Diccionarios

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).

Conjuntos

Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas. Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia, y diferencia simétrica.

Las llaves o la función set() pueden usarse para crear conjuntos. Notá que para crear un conjunto vacío tenés que usar set(), no {}; esto último crea un diccionario vacío, una estructura de datos que discutiremos en la sección siguiente.

Conjuntos

```
canasta = {'manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana'}
print(canasta)          # muestra que se removieron los duplicados
{'pera', 'manzana', 'banana', 'naranja'}

'naranja' in canasta   # verificación de pertenencia rápida
True

'yerba' in canasta
False
```

Conjuntos

```
# veamos las operaciones para las letras únicas de dos palabras
a = set('abracadabra')
b = set('alacazam')
print (a)          # letras únicas en a
{a', 'r', 'b', 'c', 'd'}

a – b              # letras en a pero no en b
{'r', 'b', 'd'}

a | b              # letras en a o en b o en ambas
{'a', 'c', 'b', 'd', 'm', 'l', 'r', 'z'}

a & b              # letras en a y en b
{'a', 'c'}

a ^ b              # letras en a o b pero no en ambos
{'b', 'd', 'm', 'l', 'r', 'z'}
```

Estructuras de control

Estructuras de control

En Python, los bloques de código dentro de cualquier estructura (control, función, objeto) se delimita mediante indentación. El código debe estar indentado con respecto a la definición de la estructura. De esta forma el código es más sencillo de leer y entender.

```
fav = “mundogeek.net”;
if fav == “mundogeek.net”:
    System.out.println(“Tienes buen gusto!”);
    System.out.println(“Gracias”);
```

Estructuras de control

En otros lenguajes de programación los bloques de código se determinan encerrándolos entre llaves (indentarlos no se trata más que de una buena práctica para facilitar la lectura del código). Por ejemplo, el código anterior expresado en Java sería algo así:

```
String fav = “mundogeek.net”;
if (fav.equals(“mundogeek.net”)){
    System.out.println(“Tienes buen gusto!”);
    System.out.println(“Gracias”);
}
```

Sentencias condicionales I

- **If**

```
if fav == “mundogeek.net”:
    print (“Tienes buen gusto!”)
print “Gracias”
```

- **if...else**

```
if fav == “mundogeek.net”:
    print (“Tienes buen gusto!”)
    print (“Gracias”)
else:
    print (“Vaya, que lástima”)
```

Sentencias condicionales II

- **if...elif...elif...else**

```
if numero < 0:
    print (“Negativo”)
elif numero > 0:
    print (“Positivo”)
else:
    print (“Cero”)
```

- **A if C else B**

```
var = “par” if (num % 2 == 0) else “impar”
```

✓ *No existe estructura switch como en Java y otros lenguajes de programación.*

Estructuras repetitivas

- **While**:

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

while True:          #Bucle infinito
    entrada = raw_input("> ")
    if entrada == "adios":
        break
    else:
        print(entrada)
```

La palabra clave break (romper) sale del bucle en el que estamos.

Estructuras repetitivas

For...in: 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)
```

Lo que hace la cabecera del bucle es obtener el siguiente elemento de la secuencia secuencia y almacenarlo en una variable de nombre elemento.

Estructuras repetitivas

Pero, ¿qué ocurre si quisiéramos utilizar el for como si estuviéramos en C o en Java, por ejemplo, para imprimir los números de 30 a 50?
No es necesario crear una lista y añadir uno a uno los números del 30 al 50. Python proporciona una función llamada range (rango) que permite generar una lista que vaya desde el primer número que le indiquemos al segundo.

```
for numero in range(30,51):      #el tope del rango (51) no se incluye
    print(numero)
```

Funciones

Funciones

En Python las funciones se declaran mediante la palabra clave **def** seguida del nombre de la función, entre paréntesis los argumentos separados por comas y acaba con los dos puntos. A partir de ahí, indentado irá el código de la función.

En la primera línea de la función nos podemos encontrar una cadena de texto llamada **docstring** (cadena de documentación).

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

mi_funcion(2,"hola)          #llamada a la función
```

También es posible definir funciones con un número variable de argumentos, o bien asignar valores por defecto a los parámetros para el caso de que no se indique ningún valor para ese parámetro al llamar a la función.

Los valores por defecto para los parámetros se definen situando un signo igual después del nombre del parámetro y a continuación el valor por defecto:

```
def imprimir(texto, veces = 1):
print veces * texto
```

Si no indicamos un valor para el segundo parámetro se imprimirá una sola vez la cadena que le pasamos como primer parámetro:

```
>>> imprimir("hola")
hola
```

si se le indica otro valor, será este el que se utilice:

```
>>> imprimir("hola", 2)
Holahola
```

Para definir funciones con un número variable de argumentos colocamos un último parámetro para la función cuyo nombre debe precederse de un signo * :

```
def varios(param1, param2, *otros):
    for val in otros:
        print val
Varios(1, 2)              #primera llamada
Varios(1, 2, 3)           #segunda llamada
Varios(1, 2, 3, 4)        #tercera llamada
```

Esta sintaxis funciona creando una tupla "*otros*" en la que se almacenan los valores de todos los parámetros extra pasados como argumento.

- En la primera llamada, "*otros*" estaría vacía.
- En la segunda llamada, "otros" valdría (3,)
- En la tercera, "otros" valdría (3, 4).

También se puede preceder el nombre del último parámetro con ** , en cuyo caso en lugar de una tupla se utilizaría un **diccionario**. Las claves de este diccionario serían los nombres de los parámetros indicados al llamar a la función y los valores del diccionario, los valores asociados a estos parámetros.

```
def varios(param1, param2, **otros):
    for i in otros.items():
        print (i)

varios(1, 2, tercero = 3)      #tercero es el índice y 3 el valor
```

Con esto terminamos todo lo relacionado con los parámetros de las funciones. Veamos por último cómo devolver valores, para lo que se utiliza la palabra clave **return** :

```
def sumar(x, y):
    return x + y

print sumar(3, 2)
```

También podríamos pasar varios valores que retornar a return.

```
def f(x, y):
    return x * 2, y * 2

a, b = f(1, 2)
```

Lo que ocurre en realidad es que Python crea una tupla al vuelo cuyos elementos son los valores a retornar, y esta única variable es la que se devuelve.

Orientación a objetos

Orientación a objetos

En Python todo son objetos. Las cadenas, por ejemplo, tienen métodos como upper() , que devuelve el texto en mayúsculas o count(sub) , que devuelve el número de veces que se encontró la cadena sub en el texto.

Clases

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.

```
class Coche:
    """Abstraccion de los objetos coche."""        #docstring
    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")
```

El método __init__ es el constructor de la clase y sirve 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, que sirve para referirse al objeto actual.

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. **Los atributos de un objeto se definen mediante el método __init__**.

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.

mi_coche = Coche(3)

Python pasa el primer argumento (la referencia al objeto que se crea, self) automáticamente.
Ahora que ya hemos creado nuestro objeto, podemos acceder a sus atributos y métodos mediante la sintaxis objeto.atributo y objeto.metodo():

```
>>> 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()
No se mueve
>>> mi_coche.arrancar()
No arranca
>>> print mi_coche.gasolina
0
```

Herencia

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”.

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
```

Como Bateria 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 sobreescribir el método __init__ para la clase Guitarra que se ejecutaría en lugar del __init__ de Instrumento.

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 usaríamos 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 usaríamos

Instrumento.__init__(self, precio)

```
class Guitarra:
    def __init__(self, precio, tipo_cuerda):
        Instrumento.__init__(self, precio)
        self.tipo_cuerda = tipo_cuerda
```

Observad que en este caso si es necesario especificar el parámetro self .

Herencia multiple

En Python 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 sobrescribirían la implementación de los métodos de las clases más a su derecha en la definición.

Herencia multiple

En el siguiente ejemplo, como Terrestre se encuentra más a la izquierda, sería la definición de desplazar de esta clase la que prevalecería, 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

La palabra polimorfismo, del griego poly morphos (varias formas), se refiere a la habilidad de objetos de distintas clases de responder al mismo mensaje. Esto se puede conseguir a través de la herencia: un objeto de una clase derivada es al mismo tiempo un objeto de la clase padre, de forma que allí donde se requiere un objeto de la clase padre también se puede utilizar uno de la clase hija.

Python, al ser de tipado dinámico, no impone restricciones a los tipos que se le pueden pasar a una función, por ejemplo, más allá de que el objeto se comporte como se espera: si se va a llamar a un método f() del objeto pasado como parámetro, por ejemplo, evidentemente el objeto tendrá que contar con ese método. Por ese motivo, a diferencia de lenguajes de tipado estático como Java o C++, el polimorfismo en Python no es de gran importancia.

Polimorfismo

En ocasiones también se utiliza el término polimorfismo para referirse a la sobrecarga de métodos, término que se define como la capacidad del lenguaje de determinar qué método ejecutar de entre varios métodos con igual nombre según el tipo o número de los parámetros que se le pasa.

En Python no existe sobrecarga de métodos (el último método sobreescribiría la implementación de los anteriores), aunque se puede conseguir un comportamiento similar recurriendo a funciones con valores por defecto para los parámetros o a la sintaxis *params o **params explicada en el punto sobre las funciones en Python.

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, como veremos al final del apartado.

```
class Ejemplo:
    def publico(self):
        print("Publico")
    def __privado(self):
        print("Privado")
ej = Ejemplo()
ej.publico()
ej.__privado()      #Se lanzará una excepción diciendo que no existe
```

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 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):
    ...
        else:
            print("Error")
    dia = property(getDia, setDia)
mi_fecha = Fecha()
mi_fecha.dia = 33
```

Métodos especiales

Ya vimos al principio del artículo el uso del método __init__ . Existen otros métodos con significados especiales, cuyos nombres siempre comienzan y terminan con dos guiones bajos. A continuación se listan algunos especialmente útiles.

__init__(self, args)
Método llamado después de crear el objeto para realizar tareas de inicialización.

__new__(cls, args)
Método exclusivo de las clases de nuevo estilo que se ejecuta antes que __init__ y que se encarga de construir y devolver el objeto en sí. Es equivalente a los constructores de C++ o Java. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es *self*, sino la propia clase: *cls*.

__del__(self)
Método llamado cuando el objeto va a ser borrado. También llamado destructor, se utiliza para realizar tareas de limpieza.

__str__(self)
Método llamado para crear una cadena de texto que represente a nuestro objeto. Se utiliza cuando usamos print para mostrar nuestro objeto o cuando usamos la función str(obj) para crear una cadena a partir de nuestro objeto.

__cmp__(self, otro)
Método llamado cuando se utilizan los operadores de comparación para comprobar si nuestro objeto es menor, mayor o igual al objeto pasado como parámetro. Debe devolver un número negativo si nuestro objeto es menor, cero si son iguales, y un número positivo si nuestro objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores < , <= , > o >= se lanzará una excepción. Si se utilizan los operadores == o != para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto (si tienen el mismo id).

__len__(self)
Método llamado para comprobar la longitud del objeto. Se utiliza, por ejemplo, cuando se llama a la función len(obj) sobre nuestro objeto. Como es de suponer, el método debe devolver la longitud del objeto.

Excepciones

Excepciones

Las excepciones son errores detectados por Python durante la ejecución del programa. Cuando el intérprete se encuentra con una situación excepcional, como el intentar dividir un número entre 0 o el intentar acceder a un archivo que no existe, este genera o lanza una excepción, informando al usuario de que existe algún problema.

Si la excepción no se captura el flujo de ejecución se interrumpe y se muestra la información asociada a la excepción en la consola de forma que el programador pueda solucionar el problema.

En Python se utiliza una construcción try - except para capturar y tratar las excepciones. El bloque try (intentar) define el fragmento de código en el que creemos que podría producirse una excepción. El bloque except (excepción) permite indicar el tratamiento que se llevará a cabo de producirse dicha excepción. El tratamiento de la excepción puede consistir en imprimir un mensaje más amigable para el usuario, registrar los errores o establecer una estrategia de resolución del problema.

En el siguiente ejemplo intentamos crear un objeto f de tipo fichero. De no existir el archivo pasado como parámetro, se lanza una excepción de tipo IOError , que capturamos gracias a nuestro try - except.

```
try:
    f = file("archivo.txt")
except:
    print("El archivo no existe")
```

Python permite utilizar varios except para un solo bloque try, de forma que podamos dar un tratamiento distinto a la excepción dependiendo del tipo de excepción de la que se trate. Esto es una buena práctica, y es tan sencillo como indicar el nombre del tipo a continuación del except .

```
try:
    num = int("3a")
    print no_existe
except NameError:
    print("La variable no existe")
except ValueError:
    print("El valor no es un numero")
```

Cuando se lanza una excepción en el bloque try , se busca en cada una de las clausulas except un manejador adecuado para el tipo de error que se produjo. En caso de que no se encuentre, se propaga la excepción.

Además podemos hacer que un mismo except sirva para tratar más de una excepción usando una tupla para listar los tipos de error que queremos que trate el bloque:

```
try:
    num = int(“3a”)
    print no_existe
except (NameError, ValueError):
    print(“Ocurrio un error”)
```

La construcción try - except puede contar además con una clausula **else**, que define un fragmento de código a ejecutar **sólo si NO se ha producido ninguna excepción** en el try.

```
try:
    num = 33
except:
    print(“Hubo un error!”)
else:
    print(“Todo esta bien”)
```

También existe una clausula **finally** que **se ejecuta siempre**, se produzca o no una excepción. Esta clausula se suele utilizar, entre otras cosas, para tareas de limpieza.

```
try:
    z = x / y
except ZeroDivisionError:
    print("Division por cero")
finally:
    print("Limpiando")
```

También es interesante comentar que como programadores podemos crear y lanzar nuestras propias excepciones. Basta crear una clase que herede de Exception o cualquiera de sus hijas y lanzarla con raise .

```
class MiError(Exception):
    def __init__(self, valor):
        self.valor = valor
    def __str__(self):
        return "Error " + str(self.valor)
try:
    if resultado > 20:
        raise MiError(33)
except MiError, e:
    print e
```

Entrada/Salida

Entrada estandar

La forma más sencilla de obtener información por parte del usuario es mediante la función **raw_input**. Esta función toma como parámetro una cadena a usar como prompt (es decir, como texto a mostrar al usuario pidiendo la entrada) y devuelve una cadena con los caracteres introducidos por el usuario hasta que pulsó la tecla Enter. Veamos un pequeño ejemplo:

```
nombre = raw_input("Como te llamas? ")
print ("Encantado, " + nombre)
```

Si necesitáramos un entero como entrada en lugar de una cadena, por ejemplo, podríamos utilizar la **función int** para convertir la cadena a entero, aunque sería conveniente tener en cuenta que puede lanzarse una excepción si lo que introduce el usuario no es un número.

```
try:
    edad = raw_input("Cuantos anyos tienes? ")
    dias = int(edad) * 365
    print ("Has vivido " + str(dias) + " dias")
except ValueError:
    print ("Eso no es un numero")
```

Salida estandar

La forma más sencilla de mostrar algo en la salida estándar es mediante el uso de la sentencia print , como hemos visto multitud de veces en ejemplos anteriores. En su forma más básica a la palabra clave print le sigue una cadena, que se mostrará en la salida estándar al ejecutarse el estamento.

```
>>> print “Hola mundo”
Hola mundo
```

Después de imprimir la cadena pasada como parámetro el puntero se sitúa en la siguiente línea de la pantalla, por lo que el print de Python funciona igual que el println de Java.

```
>>> print “Hola\n\n\tmundo”
Hola

    mundo
```

Para que la siguiente impresión se realizara en la misma línea tendríamos que colocar una coma al final de la sentencia. Comparemos el resultado de este código:

```
>>> for i in range(3):
>>> ...print i,
0 1 2
```

Con el de este otro, en el que no utiliza una coma al final de la sentencia:

```
>>> for i in range(3):
>>> ...print i
0
1
2
```

Este mecanismo de colocar una coma al final de la sentencia funciona debido a que es el símbolo que se utiliza para separar cadenas que queramos imprimir en la misma línea.

```
>>> print “Hola”, “mundo”
Hola mundo
```

Esto se diferencia del uso del operador + para concatenar las cadenas en que al utilizar las comas print introduce automáticamente un espacio para separar cada una de las cadenas. Este no es el caso al utilizar el operador + , ya que lo que le llega a print es un solo argumento: una cadena ya concatenada.

```
>>> print “Hola” + “mundo”
Holamundo
```

Además, al utilizar el operador + tendríamos que convertir antes cada argumento en una cadena de no serlo ya, ya que no es posible concatenar cadenas y otros tipos, mientras que al usar el primer método no es necesaria la conversión.

```
>>> print “Cuesta”, 3, “euros”
Cuesta 3 euros
>>> print “Cuesta” + 3 + “euros”
<type ‘exceptions.TypeError’>: cannot concatenate ‘str’ and
‘int’ objects
```

La sentencia print , o más bien las cadenas que imprime, permiten también utilizar técnicas avanzadas de formateo, de forma similar al sprintf de C. Veamos un ejemplo bastante simple:

```
print “Hola %s” % “mundo”
print “%s %s” % (“Hola”, “mundo”)
```

Lo que hace la primera línea es introducir los valores a la derecha del símbolo % (la cadena “mundo”) en las posiciones indicadas por los especificadores de conversión de la cadena a la izquierda del símbolo % , tras convertirlos al tipo adecuado. En la segunda línea, vemos cómo se puede pasar más de un valor a sustituir, por medio de una tupla.

En este ejemplo sólo tenemos un especificador de conversión: %s. Los especificadores más sencillos están formados por el símbolo % seguido de una letra que indica el tipo con el que formatear el valor.

Especificador	Formato
%s	Cadena
%d	Entero
%o	Octal
%x	Hexadecimal
%f	Real

Se puede introducir un número entre el % y el carácter que indica el tipo al que formatear, indicando el número mínimo de caracteres que queremos que ocupe la cadena. Si el tamaño de la cadena es menor que este número, se añadirán espacios a la izquierda. Si el número es negativo, los espacios se añadirán a la derecha.

```
>>> print “%10s mundo” % “Hola”
_____Hola mundo
>>> print “%-10s mundo” % “Hola”
Hola_____mundo
```

En el caso de los reales es posible indicar la precisión a utilizar precediendo la f de un punto seguido del número de decimales que queremos mostrar:

```
>>> from math import pi
>>> print “%.4f” % pi
3.1416
```

La misma sintaxis se puede utilizar para indicar el número de caracte-res de la cadena que queremos mostrar

```
>>> print “%.4s” % “hola mundo”
hola
```

Módulos y paquetes

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 creando un pequeño archivo modulo.py:

```
def mi_funcion():
    print “una funcion”
class MiClase:
    def __init__(self):
        print “una clase”
print “un modulo”
```

Para importar un módulo se utiliza la palabra clave import seguida del nombre del módulo (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 funcion” al llamar a mi_funcion , también imprimiría el texto “un modulo”, 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 y horas.

```
import os, sys, time
print time.asctime()
```

Sin duda os habréis fijado en este y el anterior ejemplo en un **detalle importante**, y es que, como vemos, es necesario preceder el nombre de los objetos que importamos de un módulo con el nombre del módulo al que pertenecen, o lo que es lo mismo, el espacio de nombres en el que se encuentran. Esto permite que no sobreescribamos accidentalmente algún otro objeto que tuviera el mismo nombre al importar otro Módulo.

Sin embargo es posible utilizar la construcción from - import para ahorrarnos el tener que indicar el nombre del módulo antes del objeto que nos interesa. De esta forma se importa el objeto o los objetos que indiquemos al espacio de nombres actual.

```
from time import asctime
print asctime()
```

Paquetes

Si los módulos sirven para organizar el código, los paquetes sirven para organizar los módulos. Los paquetes son tipos especiales de módulos (ambos son de tipo module) que permiten agrupar módulos relacionados. Mientras los módulos se corresponden a nivel físico con los archivos, los paquetes se representan mediante directorios.

Para hacer que Python trate a un directorio como un paquete es necesario crear un archivo __init__.py en dicha carpeta. En este archivo se pueden definir elementos que pertenezcan a dicho paquete, como una constante DRIVER para el paquete bbdd, aunque habitualmente se tratará de un archivo vacío. Para hacer que un cierto módulo se encuentre dentro de un paquete, basta con copiar el archivo que define el módulo al directorio del paquete.

Como los modulos, para importar paquetes también se utiliza import y from - import y el caracter . para separar paquetes, subpaquetes y módulos.

```
import paq.subpaq.modulo
paq.subpaq.modulo.func()
```