Mi primer programa en python3

Uso del intérprete

Al instalar python3 el ejecutable del intérprete lo podemos encontrar en /usr/bin/python3. Este directorio por defecto está en el PATH, por lo tanto lo podemos ejecutar directamente en el terminal. Por lo tanto para entrar en el modo interactivo, donde podemos ejecutar instrucción por instrucción interactivamente, ejecutamos:

```
$ python3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

En el modo interactivo, la última expresión impresa es asignada a la variable _.

```
>>> 4 +3
7
>>> 3 + _
10
```

Si tenemos nuestro programa en un fichero fuente (suele tener extensión py), por ejemplo programa.py,lo ejecutaremos de la siguiente manera.

```
$ python3 programa.py
```

Por defecto la codificación de nuestro código fuente es UTF-8, por lo que no debemos tener ningún problema con los caracteres utilizados en nuestro programa. Si por cualquier motivo necesitamos cambiar la codificación de los caracteres, en la primera línea de nuestro programa necesitamos poner:

```
# -*- coding: encoding -
```

Por ejemplo:

```
# -*- coding: cp-1252 -*-
```

Escribimos un programa

Un ejemplo de nuestro primer programa, podría ser este "hola mundo" un poco modificado:

```
numero = 5
if numero == 5:
        print ("Hola mundo!!!")
```

La indentación de la última línea es importante (se puede hacer con espacios o con tabulador), en python se utiliza para indicar bloques de instrucciones definidas por las estructuras de control (if, while, for, ...).

Para ejecutar este programa (guardado en hola.py):

```
$ python3 hola.py
$ Hola mundo!!!
```

Ejecución de programas usando shebang

Podemos ejecutar directamente el fichero utilizando en la primera línea el shebang, donde se indica el ejecutable que vamos a utilizar.

```
#!/usr/bin/python3
```

También podemos usar el programa env para preguntar al sistema por la ruta el intérprete de python:

```
#!/usr/bin/env python
```

Por supuesto tenemos que dar permisos de ejecución al fichero.

```
$ chmod +x hola.py
```

```
$ ./hola.py
$ Hola mundo!!!
```

Guía de estilo

Puede encontrar la guía de estilos para escribir código python en Style Guide for Python Code.

Estructura del programa

- Un programa python está formado por instrucciones que acaban en un carácter de "salto de línea".
- El punto y coma “;” se puede usar para separar varias sentencias en una misma línea, pero no se aconseja su uso.
- Una línea empieza en la primera posición, si tenemos instrucciones dentro de un bloque de una estructura de control de flujo habrá que hacer una identación.
- La identación se puede hacer con espacios y tabulaciones pero ambos tipos no se pueden mezclar. Se recomienda usar 4 espacios.
- La barra invertida "\" al final de línea se emplea para dividir una línea muy larga en dos o más líneas.
- Las expresiones entre paréntesis "()", llaves "{}" y corchetes "[]" separadas por comas "," se pueden escribir ocupando varias líneas.
- Cuando el bloque a sangrar sólo ocupa una línea ésta puede escribirse después de los dos puntos.

Comentarios

Se utiliza el carácter # para indicar los comentarios.

Palabras reservadas

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Funciones y constantes predefinidas

Funciones predefinidas

Tenemos una serie de funciones predefinidas en python3:

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()

chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Todas estas funciones y algunos elementos comunes del lenguaje están definidas en el módulo builtins.

Algunos ejemplos de funciones

- La entrada y salida de información se hacen con la función print y la función input:
- Tenemos algunas funciones matemáticas como: abs, divmod, hex, max, min,...
- Hay funciones que trabajan con caracteres y cadenas: ascii, chr, format, repr,...
- Además tenemos funciones que crean o convierten a determinados tipos de datos: int, float, str, bool, range, dict, list, ...

Iremos estudiando cada una de las funciones en las unidades correspondientes.

Constantes predefinidas

En el módulo builtins se definen las siguientes constantes:

- True y False: Valores booleanos
- None especifica que alguna variables u objeto no tiene asignado ningún tipo.

Hay alguna constante más que veremos a lo largo del curso si es necesario.

Ayuda en python

Una función fundamental cuando queremos obtener información sobre los distintos aspectos del lenguaje es help. Podemos usarla entrar en una sesión interactiva:

```
>>> help()

Welcome to Python 3.4's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.4/tutorial/.

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

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

help>

O pidiendo ayuda de una término determinado, por ejemplo:

>>> help(print)

Datos

Literales, variables y expresiones

Literales

Los literales nos permiten representar valores. Estos valores pueden ser de diferentes tipos, de esta manera tenemos diferentes tipos de literales:

Literales numéricos

- Para representar números enteros utilizamos cifras enteras (Ejemplos: 3, 12, -23). Si queremos representarlos de forma binaria comenzaremos por la secuencia 0b (Ejemplos: 0b10101, 0b1100). La representación octal la hacemos comenzando por 0o (Ejemplos: 0o377, 0o7) y por último, la representación hexadecimal se comienza por 0x (Ejemplos: 0xdeadbeef, 0xfff).

- Para los números reales utilizamos un punto para separar la parte entera de la decimal (12.3, 45.6). Podemos indicar que la parte decimal es 0, por ejemplo 10., o la parte entera es 0, por ejemplo .001, Para la representación de números muy grandes o muy pequeños podemos usar la representación exponencial (Ejemplos: 3.14e-10,1e100).

- Por último, también podemos representar números complejos, con una parte real y otra imaginaria (Ejemplo: 1+2j)

Literales cadenas

Nos permiten representar cadenas de caracteres. Para delimitar las cadenas podemos usar el carácter ' o el carácter ". También podemos utilizar la combinación "' cuando la cadena ocupa más de una línea. Ejemplos.

'hola que tal!'
"Muy bien"
"'Podemos \n
ir al cine"'

Las cadenas anteriores son del tipo str, si queremos representar una cadena de tipo byte podremos hacerlo de la siguiente manera:

b'Hola'
B"Muy bien"

Con el carácter /, podemos escapar algunos caracteres, veamos algunos ejemplos:

\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)

Variables

Una variable es un identificador que referencia a un valor. Estudiaremos más adelante que python utiliza tipado dinámico, por lo tanto no se usa el concepto de variable como almacén de información. Para que una variable referencie a un valor se utiliza el operador de asignación =.

El nombre de una variable, ha de empezar por una letra o por el carácter guión bajo, seguido de letras, números o guiones bajos. No hay que declarar la variable antes de usarla, el tipo de la variable será el mismo que el del valor al que hace referencia. Por lo tanto su tipo puede cambiar en cualquier momento:

```
>>> var = 5
>>> type(var)
<class 'int'>
>>> var = "hola"
>>> type(var)
<class 'str'>
```

Hay que tener en cuenta que python distingue entre mayúsculas y minúsculas en el nombre de una variable, pero se recomienda usar sólo minúsculas.

Definición, borrado y ámbito de variables

Como hemos comentado anteriormente para crear una variable simplemente tenemos que utilizar un operador de asignación, el más utilizado = para que referencia un valor. Si queremos borrar la variable utilizamos la instrucción del. Por ejemplo:

```
>>> a = 5
>>> a
```

```
5
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Podemos tener también variables que no tengan asignado ningún tipo de datos:

```
>>> a = None
>>> type(a)
<class 'NoneType'>
```

El ámbito de una variable se refiere a la zona del programa donde se ha definido y existe esa variable. Como primera aproximación las variables creadas dentro de funciones o clases tienen un ámbito local, es decir no existen fuera de la función o clase. Concretamos cuando estudiamos estos aspectos más profundamente.

Expresiones

Una expresión es una combinación de variables, literales, operadores, funciones y expresiones, que tras su evaluación o cálculo nos devuelven un valor de un determinado tipo.

Veamos ejemplos de expresiones:

```
a + 7
(a ** 2) + b
```

Operadores. Precedencia de operadores en python

Los operadores que podemos utilizar se clasifican según el tipo de datos con los que trabajen y podemos poner algunos ejemplos:

```
+     -     *     **     /     //     %
<<     >>     &     |     ^     ~
<     >     <=     >=     ==     !=
-=     *=     /=     //=     %=
```

Podemos clasificarlos en varios tipos:

- Operadores aritméticos
- Operadores de cadenas
- Operadores de asignación
- Operadores de comparación
- Operadores lógicos
- Operadores a nivel de bit
- Operadores de pertenencia

- Operadores de identidad

La precedencia de operadores es la siguiente:

1. Los paréntesis rompen la precedencia.
2. La potencia (**)
3. Operadores unarios (~ + -)
4. Multiplicar, dividir, módulo y división entera (* /% //)
5. Suma y resta (+ -)
6. Desplazamiento nivel de bit (>> <<)
7. Operador binario AND (&)
8. Operadores binario OR y XOR (^ |)
9. Operadores de comparación (<= < > >=)
10. Operadores de igualdad (<> == !=)
11. Operadores de asignación (= %= /= //= -= += *= **=)
12. Operadores de identidad (is, is not)
13. Operadores de pertenencia (in, in not)
14. Operadores lógicos (not, or, and)

Función eval()

La función eval() recibe una expresión como una cadena y la ejecuta.

```
>>> x=1
>>> eval("x + 1")
2
```

Tipos de datos

Podemos concretar aún más los tipos de datos (o clases) de los objetos que manejamos en el lenguaje:

- Tipos numéricos
  - Tipo entero (int)
  - Tipo real (float)
  - Tipo numérico (complex)
- Tipos booleanos (bool)
- Tipo de datos secuencia
  - Tipo lista (list)
  - Tipo tuplas (tuple)
  - Tipo rango (range)
- Tipo de datos cadenas de caracteres
  - Tipo cadena (str)
- Tipo de datos binarios
  - Tipo byte (bytes)
  - tipo bytearray (bytearray)
- Tipo de datos conjuntos
  - Tipo conjunto (set)
  - Tipo conjunto inmutable (frozenset)

- Tipo de datos iterador y generador (iter)
- Tipo de datos mapas o diccionario (dict)

En realidad todo tiene definido su tipo o clase:

- Ficheros
- Módulos
- Funciones
- Excepciones
- Clases

Función type()

La función type nos devuelve el tipo de dato de un objeto dado. Por ejemplo:

```
>>> type(5)
<class 'int'>
>>> type(5.5)
<class 'float'>
>>> type([1,2])
<class 'list'>
>>> type(int)
<class 'type'>
```

Función isinstance()

Esta función devuelve True si el objeto indicado es del tipo indicado, en caso contrario devuelve False.

```
>>> isinstance(5,int)
True
>>> isinstance(5.5,float)
True
>>> isinstance(5,list)
False
```

Trabajando con variables

Como hemos indicado anteriormente las variables en python no se declaran, se determina su tipo en tiempo de ejecución empleando una técnica que se llama **tipado dinámico**.

¿Qué es el tipado dinámico?

En python cuando asignamos una variable, se crea una referencia (puntero) al objeto creado, en ese momento se determina el tipo de la variable. Por lo tanto cada vez que asignamos de nuevo la variable puede cambiar el tipo en tiempo de ejecución.

```
>>> var = 3
>>> type(var)
<class 'int'>
>>> var = "hola"
>>> type(var)
<class 'str'>
```

Objetos inmutables y mutables

Objetos inmutables

Python procura no consumir más memoria que la necesaria. Ciertos objetos son **inmutables**, es decir, no pueden modificar su valor. El número 2 es siempre el número 2. Es un objeto inmutable. Python procura almacenar en memoria una sola vez cada valor inmutable. Si dos o más variables contienen ese valor, sus referencias apuntan a la misma zona de memoria.

Ejemplo

Para comprobar esto, vamos a utilizar la función id, que nos devuelve el identificador de la variable o el objeto en memoria.

Veamos el siguiente código:

```
>>> a = 5
```

Podemos comprobar que a hace referencia al objeto 5.

```
>>> id(5)
10771648
>>> id(a)
10771648
```

Esto es muy distinto a otros lenguajes de programación, donde una variable ocupa un espacio de memoria que almacena un valor. Desde este punto cuando asigno otro número a la variable estoy cambiando la referencia.

```
>>> a = 6
>>> id(6)
10771680
>>> id(a)
10771680
```

Las cadenas también son un objeto **inmutable**, que lo sean tiene efectos sobre las operaciones que podemos efectuar con ellas. La asignación a un elemento de una cadena, por ejemplo está prohibida:

```
>>> a = "Hola"
>>> a[0]="h"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

De los tipos de datos principales, hay que recordar que son inmutables son los números, las cadenas o las tuplas.

Objetos mutables

El caso contrario lo tenemos por ejemplo en los objetos de tipo listas, en este caso las listas son mutables. Se puede modificar un elemento de una lista.

Ejemplo

```
>>> a = [1,2]
>>> b = a
>>> id(a)
140052934508488
>>> id(b)
140052934508488
```

Como anteriormente vimos, vemos que dos variables referencia a la misma lista en memoria. Pero aquí viene la diferencia, al poder ser modificada podemos encontrar situaciones como la siguiente:

```
>>> a[0] = 5
>>> b
[5, 2]
```

Podemos ver con el id, que sigue siendo el mismo.

Cuando estudiemos las listas abordaremos este compartiendo de manera completa. De los tipos de datos principales, hay que recordar que son mutables son las listas y los diccionarios.

Operadores de identidad

Para probar esto de otra forma podemos usar los operadores de identidad:

- is: Devuelve True si dos variables u objetos están referenciando la misma posición de memoria. En caso contrario devuelve False.

- is not: Devuelve True si dos variables u objetos **no** están referenciando la misma posición de memoria. En caso contrario devuelve False.

Ejemplo

```
>>> a = 5
>>> b = a
>>> a is b
True
>>> b = b +1
>>> a is b
False
>>> b is 6
True
```

Operadores de asignación

Me permiten asignar una valor a una variable, o mejor dicho: me permiten cambiar la referencia a un nuevo objeto.

El operador principal es =:

```
>>> a = 7
>>> a
7
```

Podemos hacer diferentes operaciones con la variable y luego asignar, por ejemplo sumar y luego asignar.

```
>>> a+=2
>>> a
9
```

Otros operadores de asignación: +=, -=, *=, /=, %=, **=, //=

Asignación múltiple

En python se permiten asignaciones múltiples de esta manera:

```
>>> a, b, c = 1, 2, "hola"
```

Entrada y salida estándar

Función input

No permite leer por teclado información. Devuelve una cadena de caracteres y puede tener como argumento una cadena que se muestra en pantalla.

Ejemplos

```
>>> nombre=input("Nombre:")
Nombre:jose
>>> nombre
'jose'
>>> edad=int(input("Edad:"))
Edad:23
>>> edad
23
```

Función print

Nos permite escribir en la salida estándar. Podemos indicar varios datos a imprimir, que por defecto serán separados por un espacio (se puede indicar el separador) y por defecto se termina con un carácter salto de línea \n (también podemos indicar el carácter final). Podemos también imprimir varias cadenas de texto utilizando la concatenación.

Ejemplos

```
>>> print(1,2,3)
1 2 3
>>> print(1,2,3,sep="-")
1-2-3
>>> print(1,2,3,sep="-",end=".")
1-2-3.>>>

>>> print("Hola son las",6,"de la tarde")
Hola son las 6 de la tarde
>>> print("Hola son las "+str(6)+" de la tarde")
Hola son las 6 de la tarde
```

Formateando cadenas de caracteres

Existen dos formas de indicar el formato de impresión de las cadenas. En la documentación encontramos el estilo antiguo y el estilo nuevo.

Ejemplos del estilo antiguo

```
>>> print("%d %f %s" % (2.5,2.5,2.5))
2 2.500000 2.5

>>> print("%s %o %x"%(bin(31),31,31))
0b11111 37 1f

>>> print("El producto %s cantidad=%d precio=%.2f"%("cesta",23,13.456))
El producto cesta cantidad=23 precio=13.46
```

Función format()

Para utilizar el nuevo estilo en python3 tenemos una función format y un método format en la clase str. Vamos a ver algunos ejemplos utilizando la función format, cuando estudiemos los métodos de str lo estudiaremos con más detenimiento.

Ejemplos

```
>>> print(format(31,"b"),format(31,"o"),format(31,"x"))
11111 37 1f

>>> print(format(2.345,".2f"))
2.35
```

Tipo de datos numéricos

Python3 trabaja con tres tipos numéricos:

- Enteros (int): Representan todos los números enteros (positivos, negativos y 0), sin parte decimal. En python3 este tipo no tiene limitación de espacio.
- Reales (float): Sirve para representar los números reales, tienen una parte decimal y otra decimal. Normalmente se utiliza para su implementación un tipo double de C.
- Complejos (complex): Nos sirven para representar números complejos, con una parte real y otra imaginaria.

Como hemos visto en la unidad anterior son tipos de datos inmutables.

Ejemplos

```
>>> entero = 7
>>> type(entero)
<class 'int'>
>>> real = 7.2
>>> type (real)
<class 'float'
>>> complejo = 1+2j
>>> type(complejo)
```

<class 'complex'>

Operadores aritméticos

- +: Suma dos números
- -: Resta dos números
- *: Multiplica dos números
- /: Divide dos números, el resultado es float.
- //: División entera
- %: Módulo o resto de la división
- **: Potencia
- +, -: Operadores unarios positivo y negativo

Funciones predefinidas que trabajan con números:

- abs(x): Devuelve al valor absoluto de un número.
- divmod(x,y): Toma como parámetro dos números, y devuelve una tupla con dos valores, la división entera, y el módulo o resto de la división.
- hex(x): Devuelve una cadena con la representación hexadecimal del número que recibe como parámetro.
- oct(x): Devuelve una cadena con la representación octal del número que recibe como parámetro.
- bin(x): Devuelve una cadena con la representación binaria del número que recibe como parámetro.
- pow(x,y): Devuelve la potencia de la base x elevado al exponente y. Es similar al operador **`.
- round(x,[y]): Devuelve un número real (float) que es el redondeo del número recibido como parámetro, podemos indicar un parámetro opcional que indica el número de decimales en el redondeo.

Ejemplos

```
>>> abs(-7)
7
>>> divmod(7,2)
(3, 1)
>>> hex(255)
'0xff'
>>> oct(255)
'0o377'
>>> pow(2,3)
8
>>> round(7.567,1)
7.6
```

Operadores a nivel de bit

- x | y: x OR y

- x ^ y: x XOR y
- x & y: a AND y
- x << n: Desplazamiento a la izquierda **n** bits.
- x >> n: Desplazamiento a la derecha **n** bits.
- ~x: Devuelve los bits invertidos.

Conversión de tipos

- int(x): Convierte el valor a entero.
- float(x): Convierte el valor a float.
- complex(x): Convierte el valor a un complejo sin parte imaginaria.
- complex(x,y): Convierta el valor a un complejo, cuya parte real es x y la parte imaginaria y.

Los valores que se reciben también pueden ser cadenas de caracteres (str).

Ejemplos

```
>>> a=int(7.2)
>>> a
7
>>> type(a)
<class 'int'>
>>> a=int("345")
>>> a
345
>>> type(a)
<class 'int'>
>>> b=float(1)
>>> b
1.0
>>> type(b)
<class 'float'>
>>> b=float("1.234")
>>> b
1.234
>>> type(b)
<class 'float'>
```

Por último si queremos convertir una cadena a entero, la cadena debe estar formada por caracteres numéricos, sino es así, obtenemos un error:

```
a=int("123.3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123.3'
```

Tipo de datos booleanos

Tipo booleano

El tipo booleano o lógico se considera en python3 como un subtipo del tipo entero. Se puede representar dos valores: verdadero o false (True, False).

¿Qué valores se interpretan como FALSO?

Cuando se evalúa una expresión, hay determinados valores que se interpretan como False:

- None
- False
- Cualquier número 0. (0, 0.0, 0j)
- Cualquier secuencia vacía ([], (), '')
- Cualquier diccionario vacío ({})

Operadores booleanos

Los operadores booleanos se utilizan para operar sobre expresiones booleanas y se suelen utilizar en las estructuras de control alternativas (if, while):

- x or y: Si x es falso entonces y, sino x. Este operados sólo evalúa el segundo argumento si el primero es False.
- x and y: Si x es falso entonces x, sino y. Este operados sólo evalúa el segundo argumento si el primero es True.
- not x: Si x es falso entonces True, sino False.

Operadores de comparación

== != >= > <= <

Funciones all() y any()

- all(iterador): Recibe un iterador, por ejemplo una lista, y devuelve True si todos los elementos son verdaderos o el iterador está vacío.
- any(iterador): Recibe un iterador, por ejemplo una lista, y devuelve True si alguno de sus elemento es verdadero, sino devuelve False.

Boletín 1

1. Escribir un programa que pregunte al usuario su nombre, y luego lo salude.
2. Calcular el perímetro y área de un rectángulo dada su base y su altura.
3. Calcular el perímetro y área de un círculo dado su radio.
4. Dados dos números, mostrar la suma, resta, división y multiplicación de ambos.

5. Escribir un programa que le pida una palabra al usuario, para luego imprimirla 1000 veces, con espacios intermedios.
6. Escribir un programa que le pida una palabra al usuario, para luego imprimirla 1000 veces, con espacios intermedios.

Estructura de control: Alternativas

Si al evaluar la expresión lógica obtenemos el resultado True ejecuta un bloque de instrucciones, en otro caso ejecuta otro bloque.

Alternativas simples

```
if numero<0:
        print("Número es negativo")
```

Alternativas dobles

```
if numero<0:
        print("Número es negativo")
else:
        print("Número es positivo")
```

Alternativas múltiples

```
if numero>0:
        print("Número es negativo")
elif numero<0:
        print("Número es positivo")
else:
        print("Número es cero")
```

Expresión reducida del if

```
>>> lang="es"
>>> saludo = 'HOLA' if lang=='es' else 'HI'
>>> saludo
'HOLA'
```

Boletín 2. Alternativas

1. Realiza un programa que pida dos números ‘a’ y ‘b’ e indique si su suma es positiva, negativa o cero.
2. Escribe un programa que lea un número e indique si es par o impar.

3. Escribe un programa que pida un número entero entre uno y doce e imprima el número de días que tiene el mes correspondiente.
4. Escribe un programa que pida un nombre de usuario y una contraseña y si se ha introducido “pepe” y “asdasd” se indica “Has entrado al sistema”, sino se da un error.
5. Escribir un programa que lea un año indicar si es bisiesto. Nota: un año es bisiesto si es un número divisible por 4, pero no si es divisible por 100, excepto que también sea divisible por 400.
6. Programa que lea un carácter por teclado y compruebe si es una letra mayúscula.

Estructura de control: Repetitivas

while

La estructura while nos permite repetir un bloque de instrucciones mientras al evaluar una expresión lógica nos devuelve True. Puede tener una estructura else que se ejecutará al terminar el bucle.

Ejemplo

```
año = 2001
while año <= 2017:
        print ("Informes del Año", año)
        año += 1
else:
        print ("Hemos terminado")
```

for

La estructura for nos permite iterar los elementos de una secuencia (lista, rango, tupla, diccionario, cadena de caracteres,...). Puede tener una estructura else que se ejecutará al terminar el bucle.

Ejemplo

```
for i in range(1,10):
   print (i)
else:
   print ("Hemos terminado")
```

Instrucciones break, continue y pass

break

Termina la ejecución del bucle, además no ejecuta el bloque de instrucciones indicado por la parte else.

continue

Deja de ejecutar las restantes instrucciones del bucle y vuelve a iterar.

pass

Indica una instrucción nula, es decir no se ejecuta nada. Pero no tenemos errores de sintaxis.

Recorriendo varias secuencias. Función zip()

Con la instrucción for podemos ejecutar más de una secuencia, utilizando la función zip. Esta función crea una secuencia donde cada elemento es una tupla de los elementos de cada secuencia que toma cómo parámetro.

Ejemplo

```
>>> list(zip(range(1,4),["ana","juan","pepe"]))
[(1, 'ana'), (2, 'juan'), (3, 'pepe')]
```

Para recorrerla:

```
>>> for x,y in zip(range(1,4),["ana","juan","pepe"]):
...     print(x,y)
1 ana
2 juan
3 pepe
```

Boletín 3. Repetitivas

1. Pedir un número por teclado y mostrar la tabla de multiplicar. (hacer dos versiones una que utilice while y otra que utilice for)
2. Crea una aplicación que pida un número y calcule su factorial (El factorial de un número es el producto de todos los enteros entre 1 y el propio número y se representa por el número seguido de un signo de exclamación. Por ejemplo 5! = 1x2x3x4x5=120
3. Crea una aplicación que permita adivinar un número. En primer lugar la aplicación solicita un número entero por teclado. A continuación va pidiendo números y va

respondiendo si el número a adivinar es mayor o menor que el introducido. El programa termina cuando se acierta el número.

4. Programa que muestre la tabla de multiplicar de los números 1,2,3,4 y 5.
5. Escribe un programa que diga si un número introducido por teclado es o no primo. Un número primo es aquel que sólo es divisible entre él mismo y la unidad.

Tipo de datos secuencia

Los tipos de datos secuencia me permiten guardar una sucesión de datos de diferentes tipos. Los tipos de datos secuencias en python son:

- Las listas (list): Me permiten guardar un conjunto de datos que se pueden repetir y que pueden ser de distintos tipos. Es un tipo mutable.
- Las tuplas (tuple): Sirven para los mismo que las listas, pero en este caso es un tipo inmutable.
- Los rangos (range): Es un tipo de secuencias que nos permite crear secuencias de números. Es un tipo inmutable y se suele utilizar para realizar bucles.
- Las cadenas de caracteres (str): Me permiten guardar secuencias de caracteres. Es un tipo inmutable.
- Las secuencias de bytes (byte): Me permite guardar valores binarios representados por caracteres ASCII. Es un tipo inmutable.
- Las secuencias de bytes (bytearray): En este caso son iguales que las anteriores, pero son de tipo mutables.
- Los conjuntos (set): Me permiten guardar conjuntos de datos, en los que no se existen repeticiones. Es un tipo mutable.
- Los conjuntos (frozenset): Son iguales que los anteriores, pero son tipos inmutables.

Características principales de las secuencias

Vamos a ver distintos ejemplos partiendo de una lista, que es una secuencia mutable.

lista = [1,2,3,4,5,6]

Las secuencias se pueden recorrer

```
>>> for num in lista:
...   print(num,end="")
123456
```

Operadores de pertenencia: Se puede comprobar si un elemento pertenece o no a una secuencia con los operadores in y not in.

```
>>> 2 in lista
True
>>> 8 not in lista
True
```

Concatenación: El operador + me permite unir datos de tipos secuenciales. No se pueden concatenar secuencias de tipo range y de tipo conjunto.

```
>>> lista + [7,8,9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Repetición: El operador * me permite repetir un dato de un tipo secuencial. No se pueden repetir secuencias de tipo range y de tipo conjunto.

```
>>> lista * 2

[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Indexación: Puedo obtener el dato de una secuencia indicando la posición en la secuencia. Los conjuntos no tienen implementada esta característica.

```
>>> lista[3]
4
```

Slice (rebanada): Puedo obtener una subsecuencia de los datos de una secuencia. En los conjuntos no puedo obtener subconjuntos. Esta característica la estudiaremos más detenidamente en la unidad que estudiemos las listas.

```
>>> lista[2:4]
[3, 4]
>>> lista[1:4:2]
[2, 4]
```

Con la función **len** puedo obtener el tamaño de la secuencia, es decir el número de elementos que contiene.

```
>>> len(lista)
6
```

Con las funciones **max** y **min** puedo obtener el valor máximo y mínimo de una secuencia.

```
>>> max(lista)
6
>>> min(lista)
1
```

Además en los tipos mutables puedo realizar las siguientes operaciones:

Puedo modificar un dato de la secuencia indicando su posición.

```
>>> lista[2]=99
>>> lista
[1, 2, 99, 4, 5, 6]
```

Puedo modificar un subconjunto de elementos de la secuencia haciendo **slice**.

```
>>> lista[2:4]=[8,9,10]
>>> lista
[1, 2, 8, 9, 10, 5, 6]
```

Puedo borrar un subconjunto de elementos con la instrucción **del**.

```
>>> del lista[5:]

>>> lista
[1, 2, 8, 9, 10]
```

Puedo actualizar la secuencia con el operador ***=**

```
>>> lista*=2

>>> lista
[1, 2, 8, 9, 10, 1, 2, 8, 9, 10]
```

Tipo de datos secuencia: listas

Las listas (list) me permiten guardar un conjunto de datos que se pueden repetir y que pueden ser de distintos tipos. Es un tipo mutable.

Construcción de una lista

Para crear una lista puedo usar varias formas:

Con los caracteres [y]:

```
>>> lista1 = []
>>> lista2 = ["a",1,True]
```

Utilizando el constructor list, que toma como parámetro un dato de algún tipo secuencia.

```
>>> lista3 = list()
>>> lista4 = list("hola")
>>> lista4
['h', 'o', 'l', 'a']
```

Operaciones básicas con listas

Como veíamos en el apartado "Tipo de datos secuencia" podemos realizar las siguientes operaciones:

- Las secuencias se pueden recorrer.
- Operadores de pertenencia: in y not in.
- Concatenación: +
- Repetición: *

Indexación: Cada elemento tiene un índice, empezamos a contar por el elemento en el índice 0. Si intentamos acceder a un índice que corresponda a un elemento que no existe obtenemos una excepción IndexError.

```
>>> lista1[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
IndexError: list index out of range
```

Se pueden utilizar índices negativos:

```
>>> lista[-1]
6
```

Slice: Veamos como se puede utilizar

```
lista[start:end]  # Elementos desde la posición start hasta end-1
lista[start:]     # Elementos desde la posición start hasta el final
lista[:end]       # Elementos desde el principio hasta la posición end-1
lista[:]          # Todos Los elementos
lista[start:end:step] # Igual que el anterior pero dando step saltos.
```

- Se pueden utilizar también índices negativos, por ejemplo: lista[::-1]

Funciones predefinidas que trabajan con listas

```
>>> lista1 = [20,40,10,40,50]
>>> len(lista1)
5
>>> max(lista1)
50
>>> min(lista1)
10
>>> sum(lista1)
150
>>> sorted(lista1)
[10, 20, 30, 40, 50]
>>> sorted(lista1,reverse=True)
[50, 40, 30, 20, 10]
```

Veamos con más detenimiento la función enumerate: que recibe una secuencia y devuelve un objeto enumerado como tuplas:

```
>>> seasons = ['Primavera', 'Verano', 'Otoño', 'Invierno']
>>> list(enumerate(seasons))
[(0, 'Primavera'), (1, 'Verano'), (2, 'Otoño'), (3, 'Invierno')]
>>> list(enumerate(seasons, start=1))
[(1, 'Primavera'), (2, 'Verano'), (3, 'Otoño'), (4, 'Invierno')]
```

Las listas son mutables

Como hemos indicado anteriormente las listas son un tipo de datos mutable. Eso tiene para nosotros varias consecuencias, por ejemplo podemos obtener resultados como se los que se muestran a continuación:

```
>>> lista1 = [1,2,3]
>>> lista1[2]=4
>>> lista1
[1, 2, 4]
>>> del lista1[2]
>>> lista1
[1, 2]
```

```
>>> lista1 = [1,2,3]
>>> lista2 = lista1
```

```
>>> lista1[1] = 10
>>> lista2
[1, 10, 3]
```

¿Cómo se copian las listas?

Por lo tanto si queremos copiar una lista en otra podemos hacerlo de varias formas:

```
>>> lista1 = [1,2,3]
>>> lista2=lista1[:]
>>> lista1[1] = 10
>>> lista2
[1, 2, 3]

>>> lista2 = list(lista1)

>>> lista2 = lista1.copy()
```

Listas multidimensionales

A la hora de definir las listas hemos indicado que podemos guardar en ellas datos de cualquier tipo, y evidentemente podemos guardar listas dentro de listas.

```
>>> tabla = [[1,2,3],[4,5,6],[7,8,9]]
>>> tabla[1][1]
5

>>> for fila in tabla:
...   for elem in fila:
...     print(elem,end="")
...   print()

123
456
789
```

Métodos principales de listas

Cuando creamos una lista estamos creando un objeto de la clase list, que tiene definido un conjunto de métodos:

lista.append lista.copy lista.extend lista.insert lista.remove lista.sort
lista.clear lista.count lista.index lista.pop lista.reverse

Métodos de inserción: append, extend, insert

```
>>> lista = [1,2,3]
>>> lista.append(4)
>>> lista
[1, 2, 3, 4]

>>> lista2 = [5,6]
>>> lista.extend(lista2)
>>> lista
[1, 2, 3, 4, 5, 6]

>>> lista.insert(1,100)
>>> lista
[1, 100, 2, 3, 4, 5, 6]
```

Métodos de eliminación: pop, remove

```
>>> lista.pop()
6
>>> lista
[1, 100, 2, 3, 4, 5]

>>> lista.pop(1)
100
>>> lista
[1, 2, 3, 4, 5]

>>> lista.remove(3)
>>> lista
[1, 2, 4, 5]
```

Métodos de ordenación: reverse, sort,

```
>>> lista.reverse()
>>> lista
[5, 4, 2, 1]

>>> lista.sort()
>>> lista
[1, 2, 4, 5]

>>> lista.sort(reverse=True)
```

```
>>> lista
[5, 4, 2, 1]

>>> lista=["hola","que","tal","Hola","Que","Tal"]
>>> lista.sort()
>>> lista
['Hola', 'Que', 'Tal', 'hola', 'que', 'tal']
>>> lista=["hola","que","tal","Hola","Que","Tal"]
>>> lista.sort(key=str.lower)
>>> lista
['hola', 'Hola', 'que', 'Que', 'tal', 'Tal']
```

Métodos de búsqueda: count, index

```
>>> lista.count(5)
1

>>> lista.append(5)
>>> lista
[5, 4, 2, 1, 5]
>>> lista.index(5)
0
>>> lista.index(5,1)
4
>>> lista.index(5,1,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in list
```

Método de copia: copy

```
>>> lista2 = lista1.copy()
```

Ejercicios de listas

1. Lee por teclado números y guárdalo en una lista, el proceso finaliza cuando metamos un número negativo. Muestra el máximo de los números guardados en la lista, muestra los números pares
2. Realizar un programa que, dada una lista, devuelve una nueva lista cuyo contenido sea igual a la original pero invertida. Así, dada la lista [‘Di’, ‘buen’, ‘día’, ‘a’, ‘papa’], deberá devolver [‘papa’, ‘a’, ‘día’, ‘buen’, ‘Di’].

3. Dada una lista de cadenas, pide una cadena por teclado e indica si está en la lista, indica cuántas veces aparece en la lista, lee otra cadena y sustituye la primera por la segunda en la lista, y por último borra la cadena de la lista
4. Dado una lista, hacer un programa que indique si está ordenada o no.

Operaciones avanzadas con secuencias

Las funciones que vamos a estudiar en esta unidad nos acercan al paradigma de la programación funcional que también nos ofrece python. La programación funcional es un paradigma de programación declarativa basado en el uso de funciones matemáticas, en contraste con la programación imperativa, que enfatiza los cambios de estado mediante la mutación de variables.

Función map

map(funcion,secuencia): Ejecuta la función enviada por parámetro sobre cada uno de los elementos de la secuencia.

Ejemplo

```
>>> items = [1, 2, 3, 4, 5]
>>> def sqr(x): return x ** 2
>>> list(map(sqr, items))
[1, 4, 9, 16, 25]
```

Función filter

filter(funcion,secuencia): Devuelve una secuencia con los elementos de la secuencia envíada por parámetro que devuelvan True al aplicarle la función envíada también como parámetro.

Ejemplo

```
>>> lista = [1,2,3,4,5]
>>> def par(x): return x % 2==0
>>> list(filter(par,lista))
```

Función reduce

reduce(funcion,secuencia): Devuelve un único valor que es el resultado de aplicar la función á los elementos de la secuencia.

Ejemplo

```
>>> from functools import reduce
>>> lista = [1,2,3,4,5]
>>> def add(x,y): return x + y
>>> reduce(add,lista)
15
```

list comprehension

list comprehension nos proporciona una alternativa para la creación de listas. Es parecida a la función map, pero mientras map ejecuta una función por cada elemento de la secuencia, con esta técnica se aplica una expresión.

Ejemplo

```
>>> [x ** 3 for x in [1,2,3,4,5]]
[1, 8, 27, 64, 125]

>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]

>>> [x + y for x in [1,2,3] for y in [4,5,6]]
[5, 6, 7, 6, 7, 8, 7, 8, 9]
```

Tipo de datos secuencia: Tuplas

Las tuplas (tuple): Sirven para los mismo que las listas (me permiten guardar un conjunto de datos que se pueden repetir y que pueden ser de distintos tipos), pero en este caso es un tipo inmutable.

Construcción de una tupla

Para crear una lista puedo usar varias formas:

Con los caracteres (y):

```
>>> tupla1 = ()
>>> tupla2 = ("a",1,True)
```

Utilizando el constructor tuple, que toma como parámetro un dato de algún tipo secuencia.

```
>>> tupla3=tuple()
```

```
>>> tuple4=tuple([1,2,3])
```

Empaquetado y desempaquetado de tuplas

Si a una variable se le asigna una secuencia de valores separados por comas, el valor de esa variable será la tupla formada por todos los valores asignados.

```
>>> tuple = 1,2,3
>>> tuple
(1, 2, 3)
```

Si se tiene una tupla de longitud k, se puede asignar la tupla a k variables distintas y en cada variable quedará una de las componentes de la tupla.

```
>>> a,b,c=tuple
>>> a
1
```

Operaciones básicas con tuplas

En las tuplas se pueden realizar las siguientes operaciones:

- Las tuplas se pueden recorrer.
- Operadores de pertenencia: in y not in.
- Concatenación: +
- Repetición: *
- Indexación
- Slice

Entre las funciones definidas podemos usar: len, max, min, sum, sorted.

Las tuplas son inmutables

```
>>> tupla = (1,2,3)
>>> tupla[1]=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Métodos principales

Métodos de búsqueda: count, index

```
>>> tupla = (1,2,3,4,1,2,3)
>>> tupla.count(1)
2

>>> tupla.index(2)
1
>>> tupla.index(2,2)
5
```

Tipo de datos secuencia: Rangos

Los rangos (range): Es un tipo de secuencias que nos permite crear secuencias de números. Es un tipo inmutable y se suele utilizar para realizar bucles.

Definición de un rango. Constructor range

Al crear un rango (secuencia de números) obtenemos un objeto que es de la clase range:

```
>>> rango = range(0,10,2)
>>> type(rango)
<class 'range'>
```

Veamos algunos ejemplos, convirtiendo el rango en lista para ver la secuencia:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Recorrido de un rango

Los rangos se suelen usar para ser recorrido, cuando tengo que crear un bucle cuyo número de iteraciones lo se de antemanos puedo usar una estructura como esta:

```
>>> for i in range(11):
...     print(i,end=" ")
0 1 2 3 4 5 6 7 8 9 10
```

Operaciones básicas con range

En las tuplas se pueden realizar las siguientes operaciones:

- Los rangos se pueden recorrer.
- Operadores de pertenencia: in y not in.
- Indexación
- Slice

Entre las funciones definidas podemos usar: len, max, min, sum, sorted.

Además un objeto range posee tres atributos que nos almacenan el comienzo, final e intervalo del rango:

```
>>> rango = range(1,11,2)
>>> rango.start
1
>>> rango.stop
11
>>> rango.step
2
```

Codificación de caracteres

Introducción a la codificación de caracteres

ascii

En los principios de la informática los ordenadores se diseñaron para utilizar sólo caracteres ingleses, por lo tanto se creó una codificación de caracteres, llamada ascii (American Standard Code for Information Interchange) que utiliza 7 bits para codificar los 128 caracteres necesarios en el alfabeto inglés. Posteriormente se extendió esta codificación para incluir caracteres no ingleses. Al utilizar 8 bits se pueden representar 256 caracteres. De esta forma para codificar el alfabeto latino aparece la codificación ISO-8859-1 o Latín 1.

Unicode

La codificación unicode nos permite representar todos los caracteres de todos los alfabetos del mundo, en realidad permite representar más de un millón de caracteres, ya que utiliza 32 bits para su representación, pero en la realidad sólo se definen unos 110.000 caracteres.

UTF-8

UTF-8 es un sistema de codificación de longitud variable para Unicode. Esto significa que los caracteres pueden utilizar diferente número de bytes.

La codificación de caracteres en python3

En Python 3.x las cadenas de caracteres pueden ser de tres tipos: Unicode, Byte y Bytearray.

- El tipo unicode permite caracteres de múltiples lenguajes y cada carácter en una cadena tendrá un valor inmutable.
- El tipo byte sólo permitirá caracteres ASCII y los caracteres son también inmutables.
- El tipo bytearray es como el tipo byte pero, en este caso, los caracteres de una cadena si son mutables.

Algo que debe entenderse (e insiste Mark Pilgrim en su libro *Dive into Python*) es que "los bytes no son caracteres, los bytes son bytes; un carácter es en realidad una abstracción; y una cadena de caracteres es una sucesión de abstracciones".

Funciones chr() y ord()

chr(i): Nos devuelve el carácter Unicode que representa el código i.

```
>>> chr(97)
'a'
>>> chr(1004)
'Ϭ'
```

-

ord(c): recibe un carácter c y devuelve el código unicode correspondiente.

```
>>> ord("a")
97
>>> ord("Ϭ")
1004
```

Tipo de datos cadenas de caracteres

- Las cadenas de caracteres (str): Me permiten guardar secuencias de caracteres. Es un tipo inmutable. Como hemos comentado las cadenas de caracteres en python3 están codificadas con Unicode.

Definición de cadenas. Constructor str

Podemos definir una cadena de caracteres de distintas formas:

```
>>> cad1 = "Hola"
>>> cad2 = '¿Qué tal?'
>>> cad3 = '''Hola,
        que tal?'''
```

También podemos crear cadenas con el constructor str a partir de otros tipos de datos.

```
>>> cad1=str(1)
>>> cad2=str(2.45)
>>> cad3=str([1,2,3])
```

Operaciones básicas con cadenas de caracteres

Como veíamos en el apartado "Tipo de datos secuencia" podemos realizar las siguientes operaciones:

- Las cadenas se pueden recorrer.
- Operadores de pertenencia: in y not in.
- Concatenación: +
- Repetición: *
- Indexación
- Slice

Entre las funciones definidas podemos usar: len, max, min, sorted.

Las cadenas son inmutables

```
>>> cad = "Hola que tal?"
>>> cad[4]="."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Comparación de cadenas

Las cadenas se comparan carácter a carácter, en el momento en que dos caracteres no son iguales se compara alfabéticamente (es decir, se convierte a código unicode y se comparan).

Ejemplos

```
>>> "a">"A"
True
>>> ord("a")
97
>>> ord("A")
65

>>> "informatica">"informacion"
True

>>> "abcde">"abcdef"
False
```

Funciones repr, ascii, bin

repr(objeto): Devuelve una cadena de caracteres que representa la información de un objeto.

```
>>> repr(range(10))
'range(0, 10)'
>>> repr("piña")
"'piña'"
```

La cadena devuelta por repr() debería ser aquella que, pasada a eval(), devuelve el mismo objeto.

```
>>> type(eval(repr(range(10))))
<class 'range'>
```

ascii(objeto): Devuelve también la representación en cadena de un objeto pero en este caso muestra los caracteres con un código de escape . Por ejemplo en ascii (Latin1) la á se presenta con \xe1.

```
>>> ascii("á")
"'\\xe1'"
>>> ascii("piña")
"'pi\\xf1a'"
```

bin(numero): Devuelve una cadena de caracteres que corresponde a la representación binaria del número recibido.

```
>>> bin(213)
'0b11010101'
```

Métodos principales de cadenas

Cuando creamos una cadena de caracteres estamos creando un objeto de la clase str, que tiene definido un conjunto de métodos:

cadena.capitalize cadena.isalnum cadena.join cadena.rsplit
cadena.casefold cadena.isalpha cadena.ljust cadena.rstrip
cadena.center cadena.isdecimal cadena.lower cadena.split
cadena.count cadena.isdigit cadena.lstrip cadena.splitlines
cadena.encode cadena.isidentifier cadena.maketrans cadena.startswith
cadena.endswith cadena.islower cadena.partition cadena.strip
cadena.expandtabs cadena.isnumeric cadena.replace cadena.swapcase
cadena.find cadena.isprintable cadena.rfind cadena.title
cadena.format cadena.isspace cadena.rindex cadena.translate
cadena.format_map cadena.istitle cadena.rjust cadena.upper
cadena.index cadena.isupper cadena.rpartition cadena.zfill

Métodos de formato

```
>>> cad = "hola, como estás?"
>>> print(cad.capitalize())
Hola, como estás?

>>> cad = "Hola Mundo"
>>> print(cad.lower())
hola mundo

>>> cad = "hola mundo"
>>> print(cad.upper())
HOLA MUNDO

>>> cad = "Hola Mundo"
>>> print(cad.swapcase())
hOLA mUNDO

>>> cad = "hola mundo"
>>> print(cad.title())
```

Hola Mundo

```
>>> print(cad.center(50))
                  hola mundo
>>> print(cad.center(50,"="))
====================hola mundo====================

>>> print(cad.ljust(50,"="))
hola mundo========================================
>>> print(cad.rjust(50,"="))
========================================hola mundo

>>> num = 123
>>> print(str(num).zfill(12))
000000000123
```

Métodos de búsqueda

```
>>> cad = "bienvenido a mi aplicación"
>>> cad.count("a")
3
>>> cad.count("a",16)
2
>>> cad.count("a",10,16)
1

>>> cad.find("mi")
13
>>> cad.find("hola")
-1

>>> cad.rfind("a")
21
```

El método index() y rindex() son similares a los anteriores pero provocan una excepción ValueError cuando no encuentra la subcadena.

Métodos de validación

```
>>> cad.startswith("b")
True
>>> cad.startswith("m")
False
>>> cad.startswith("m",13)
True
```

```
>>> cad.endswith("ción")
True
>>> cad.endswith("ción",0,10)
False
>>> cad.endswith("nido",0,10)
True
```

Otras funciones de validación: isalnum(), isalpha(), isdigit(), islower(), isupper(), isspace(), istitle(),...

Métodos de sustitución

format

En la unidad **"Entrada y salida estándar"** ya estuvimos introduciendo el concepto de formateo de las cadenas. Estuvimos viendo que hay dos métodos y vimos algunos ejemplos del *nuevo estilo* con la función predefinida format().

El uso del estilo nuevo es actualmente el recomendado (puedes obtener más información y ejemplos en algunos de estos enlaces: enlace1 y enlace2) y obtiene toda su potencialidad usando el método format() de las cadenas. Veamos algunos ejemplos:

```
>>> '{} {}'.format("a", "b")
'a b'
>>> '{1} {0}'.format("a", "b")
'b a'
>>> 'Coordenadas: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordenadas: 37.24N, -115.81W'
>>> '{0:b} {1:x} {2:.2f}'.format(123, 223,12.2345)
'1111011 df 12.23'
>>> '{:^10}'.format('test')
'   test   '
```

Otros métodos de sustitución

```
>>> buscar = "nombre apellido"
>>> reemplazar_por = "Juan Pérez"
>>> print ("Estimado Sr. nombre apellido:".replace(buscar, reemplazar_por))
Estimado Sr. Juan Pérez:

>>> cadena = "   www.eugeniabahit.com   "
>>> print(cadena.strip())
www.eugeniabahit.com
>>> cadena="00000000123000000000"
>>> print(cadena.strip("0"))
123
```

De forma similar lstrip(["caracter"]) y rstrip(["caracter"]).

Métodos de unión y división

```
>>> formato_numero_factura = ("Nº 0000-0", "-0000 (ID: ", ")"
>>> print("275".join(formato_numero_factura))
Nº 0000-0275-0000 (ID: 275)

>>> hora = "12:23"
>>> print(hora.rpartition(":"))
('12', ':', '23')
>>> print(hora.partition(":"))
('12', ':', '23')
>>> hora = "12:23:12"
>>> print(hora.partition(":"))
('12', ':', '23:12')
>>> print(hora.split(":"))
['12', '23', '12']
>>> print(hora.rpartition(":"))
('12:23', ':', '12')
>>> print(hora.rsplit(":",1))
['12:23', '12']


>>> texto = "Linea 1\nLinea 2\nLinea 3"
>>> print(texto.splitlines())
['Linea 1', 'Linea 2', 'Linea 3']
```

Ejercicios de cadenas

1. Crear un programa que lea por teclado una cadena y un carácter, e inserte el carácter entre cada letra de la cadena. Ej: separar y , debería devolver s,e,p,a,r,a,r
2. Crear un programa que lea por teclado una cadena y un carácter, y reemplace todos los dígitos en la cadena por el carácter. Ej: su clave es: 1540 y X debería devolver su clave es: XXXX
3. Crea un programa python que lea una cadena de caracteres y muestre la siguiente información:
    - La primera letra de cada palabra. Por ejemplo, si recibe Universal Serial Bus debe devolver USB.
    - Dicha cadena con la primera letra de cada palabra en mayúsculas. Por ejemplo, si recibe república argentina debe devolver República Argentina.
    - Las palabras que comiencen con la letra A. Por ejemplo, si recibe Antes de ayer debe devolver Antes ayer.

4. Escribir funciones que dadas dos cadenas de caracteres:
    - Indique si la segunda cadena es una subcadena de la primera. Por ejemplo, cadena es una subcadena de subcadena.
    - Devuelva la que sea anterior en orden alfabético. Por ejemplo, si recibe kde y gnome debe devolver gnome.
5. Escribir un programa python que dado una palabra diga si es un palíndromo. Un palíndromo es una palabra, número o frase que se lee igual hacia adelante que hacia atrás. Ejemplo: reconocer

Tipo de datos binarios: bytes, bytearray

Bytes

El tipo bytes es una secuencia inmutable de bytes. Sólo admiten caracteres ASCII. También se pueden representar los bytes mediante números enteros cuyo valores deben cumplir 0 <= x < 256.

Definición de bytes. Constructor bytes

Podemos definir un tipo bytes de distintas formas:

```
>>> byte1 = b"Hola"
>>> byte2 = b'¿Qué tal?'
>>> byte3 = b'''Hola,
        que tal?'''
```

También podemos crear cadenas con el constructor bytes a partir de otros tipos de datos.

```
>>> byte1=bytes(10)
>>> byte1
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> byte2=bytes(range(10))
>>> byte2
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t'
>>> byte3=bytes.fromhex('2Ef0 F1f2')
>>> byte3
b'.\xf0\xf1\xf2'
```

Bytearray

El tipo bytearray es un tipo mutable de bytes.

Definición de bytearray. Constructor bytearray

```
>>> ba1=bytearray()
>>> ba1
bytearray(b'')
>>> ba2=bytearray(10)
>>> ba2
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> ba3=bytearray(range(10))
>>> ba3
bytearray(b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t')
>>> ba4=bytearray(b"hola")
>>> ba4
bytearray(b'hola')
>>> ba5=bytearray.fromhex('2Ef0 F1f2')
>>> ba5
bytearray(b'.\xf0\xf1\xf2')
```

Operaciones básicas con bytes y bytearray

Como veíamos en el apartado "Tipo de datos secuencia" podemos realizar las siguientes operaciones:

- Recorrido
- Operadores de pertenencia: in y not in.
- Concatenación: +
- Repetición: *
- Indexación
- Slice

Entre las funciones definidas podemos usar: len, max, min, sum, sorted.

Los bytes son inmutables, los bytearray son mutables

```
>>> byte=b"hola"
>>> byte[2]=b'g'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment

>>> ba1=bytearray(b'hola')
>>> ba1[2]=123
>>> ba1
bytearray(b'ho{a')
>>> del ba1[3]
>>> ba1
bytearray(b'ho{')
```

Métodos de bytes y bytearray

byte1.capitalize byte1.index byte1.join byte1.rindex byte1.strip
byte1.center byte1.isalnum byte1.ljust byte1.rjust byte1.swapcase
byte1.count byte1.isalpha byte1.lower byte1.rpartition byte1.title
byte1.decode byte1.isdigit byte1.lstrip byte1.rsplit byte1.translate
byte1.endswith byte1.islower byte1.maketrans byte1.rstrip byte1.upper
byte1.expandtabs byte1.isspace byte1.partition byte1.split byte1.zfill
byte1.find byte1.istitle byte1.replace byte1.splitlines
byte1.fromhex byte1.isupper byte1.rfind byte1.startswith

bytearray1.append bytearray1.index bytearray1.lstrip bytearray1.rstrip
bytearray1.capitalize bytearray1.insert bytearray1.maketrans bytearray1.split
bytearray1.center bytearray1.isalnum bytearray1.partition bytearray1.splitlines
bytearray1.clear bytearray1.isalpha bytearray1.pop bytearray1.startswith
bytearray1.copy bytearray1.isdigit bytearray1.remove bytearray1.strip
bytearray1.count bytearray1.islower bytearray1.replace bytearray1.swapcase
bytearray1.decode bytearray1.isspace bytearray1.reverse bytearray1.title
bytearray1.endswith bytearray1.istitle bytearray1.rfind bytearray1.translate
bytearray1.expandtabs bytearray1.isupper bytearray1.rindex bytearray1.upper
bytearray1.extend bytearray1.join bytearray1.rjust bytearray1.zfill
bytearray1.find bytearray1.ljust bytearray1.rpartition
bytearray1.fromhex bytearray1.lower bytearray1.rsplit

Si nos fijamos la mayoría de los métodos en el caso de los bytes son los de las cadenas de caracteres, y en los bytearray encontramos también métodos propios de las listas.

Métodos encode y decode

Los caracteres cuyo código es mayor que 256 no se pueden usar para representar los bytes, sin embargo sí podemos indicar una codificación de caracteres determinada para que ese carácter se convierta en un conjunto de bytes.

```
>>> byte1=b'piña'
  File "<stdin>", line 1
SyntaxError: bytes can only contain ASCII literal characters.
>>> byte1=bytes('piña',"utf-8")
>>> byte1
b'pi\xc3\xb1a'
>>> len(byte1)
5
>>> byte1=bytes('piña',"latin1")
>>> byte1
b'pi\xf1a'
```

Podemos también convertir una cadena unicode a bytes utilizando el método encode:

```
>>> cad="piña"
>>> byte1=cad.encode("utf-8")
>>> byte1
b'pi\xc3\xb1a'
```

Para hacer la función inversa, convertir de bytes a unicode utilizamos el método decode:

```
>>> byte1.decode("utf-8")
'piña'
```

El problema lo tenemos si hemos codificado utilizando un código e intentamos decodificar usando otro.

```
>>> byte1=bytes('piña',"latin1")
>>> byte1.decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in position 2: invalid continuation byte
>>> byte1.decode("utf-8","ignore")
'pia'
>>> byte1.decode("utf-8","replace")
'pi�a'
```

Tipo de datos conjuntos: set, frozenset

set

Los conjuntos (set): Me permiten guardar conjuntos (desordenados) de datos (a los que se puede calcular una función hash), en los que no existen repeticiones. Es un tipo de datos mutable.

Normalmente se usan para comprobar si existe un elemento en el conjunto, eliminar duplicados y cálculos matemáticos, como la intersección, unión, diferencia,...

Definición de set. Constructor set

Podemos definir un tipo set de distintas formas:

```
>>> set1 = set()
>>> set1
set()
>>> set2=set([1,1,2,2,3,3])
```

```
>>> set2
{1, 2, 3}
>>> set3={1,2,3}
>>> set3
{1, 2, 3}
```

Frozenset

El tipo frozenset es un tipo inmutable de conjuntos.

Definición de frozenset. Constructor frozenset

```
>>> fs1=frozenset()
>>> fs1
frozenset()
>>> fs2=frozenset([1,1,2,2,3,3])
>>> fs2
frozenset({1, 2, 3})
```

Operaciones básicas con set y frozenset

De las operaciones que estudiamos en el apartado "Tipo de datos secuencia" los conjuntos sólo aceptan las siguientes:

- Recorrido
- Operadores de pertenencia: in y not in.

Entre las funciones definidas podemos usar: len, max, min, sorted.

Los set son mutables, los frozenset son inmutables

```
>>> set1={1,2,3}
>>> set1.add(4)
>>> set1
{1, 2, 3, 4}
>>> set1.remove(2)
>>> set1
{1, 3, 4}
```

El tipo frozenset es inmutable por lo tanto no posee los métodos add y remove.

Métodos de set y frozenset

set1.add set1.issubset

```
set1.clear                    set1.issuperset
set1.copy                     set1.pop
set1.difference               set1.remove
set1.difference_update        set1.symmetric_difference
set1.discard                  set1.symmetric_difference_update
set1.intersection             set1.union
set1.intersection_update      set1.update
set1.isdisjoint
```

Veamos algunos métodos, partiendo siempre de estos dos conjuntos:

```
>>> set1={1,2,3}
>>> set2={2,3,4}

>>> set1.difference(set2)
{1}
>>> set1.difference_update(set2)
>>> set1
{1}

>>> set1.symmetric_difference(set2)
{1, 4}
>>> set1.symmetric_difference_update(set2)
>>> set1
{1, 4}

>>> set1.intersection(set2)
{2, 3}
>>> set1.intersection_update(set2)
>>> set1
{2, 3}

>>> set1.union(set2)
{1, 2, 3, 4}
>>> set1.update(set2)
>>> set1
{1, 2, 3, 4}
```

Veamos los métodos de añadir y eliminar elementos:

```
>>> set1 = set()
>>> set1.add(1)
>>> set1.add(2)
>>> set1
{1, 2}
>>> set1.discard(3)
```

```
>>> set1.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
>>> set1.pop()
1
>>> set1
{2}
```

Y los métodos de comprobación:

```
>>> set1 = {1,2,3}
>>> set2 = {1,2,3,4}
>>> set1.isdisjoint(set2)
False
>>> set1.issubset(set2)
True
>>> set1.issuperset(set2)
False
>>> set2.issuperset(set1)
True
```

Por último los métodos de frozenset:

```
fset1.copy              fset1.isdisjoint        fset1.symmetric_difference
fset1.difference        fset1.issubset          fset1.union
fset1.intersection      fset1.issuperset
```

Tipo de datos: iterador y generador

Iterador

Un objeto iterable es aquel que puede devolver un iterador. Normalmente las colecciones que hemos estudiado son iterables. Un iterador me permite recorrer los elementos del objeto iterable.

Definición de iterador. Constructor iter

```
>>> iter1 = iter([1,2,3])
>>> type(iter1)
<class 'list_iterator'>
>>> iter2 = iter("hola")
>>> type(iter2)
```

<class 'str_iterator'>

Función next(), reversed()

Para recorrer el iterador, utilizamos la función next():

```
>>> next(iter1)
1
>>> next(iter1)
2
>>> next(iter1)
3
>>> next(iter1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

La función reversed() devuelve un iterador con los elementos invertidos, desde el último al primero.

```
>>> iter2 = reversed([1,2,3])
>>> next(iter2)
3
>>> next(iter2)
2
>>> next(iter2)
1
>>> next(iter2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

El módulo itertools

El módulo itertools contiene distintas funciones que nos devuelven iteradores.

Veamos algunos ejemplos:

count(): Devuelve un iterador infinito.

```
>>> from itertools import count
>>> counter = count(start=13)
>>> next(counter)
13
```

```
>>> next(counter)
14
```

cycle(): devuelve una secuencia infinita.

```
>>> from itertools import cycle
>>> colors = cycle(['red', 'white', 'blue'])
>>> next(colors)
'red'
>>> next(colors)
'white'
>>> next(colors)
'blue'
>>> next(colors)
'red'
```

islice(): Retorna un iterador finito.

```
>>> from itertools import islice
>>> limited = islice(colors, 0, 4)
>>> for x in limited:
...   print(x)
white
blue
red
white
```

Generadores

Un generador es un tipo concreto de iterador. Es una función que permite obtener sus resultados paso a paso. Por ejemplo, hacer una función que cada vez que la llamemos nos de el próximo número par. Tenemos dos maneras de crear generadores:

1. Realizar una función que devuelva los valores con la palabra reservada yield. Lo veremos con profundidad cuando estudiemos las funciones.

Utilizando la sintaxis de las "list comprehension". Por ejemplo:

```
>>> iter1 = (x for x in range(10) if x % 2==0)
>>> next(iter1)
0
>>> next(iter1)
2
>>> next(iter1)
4
```

Tipo de datos mapa: diccionario

Los diccionarios son tipos de datos que nos permiten guardar valores, a los que se puede acceder por medio de una clave. Son tipos de datos mutables y los campos no tienen asignado orden.

Definición de diccionarios. Constructor dict

```
>>> a = dict(one=1, two=2, three=3)

>>> b = {'one': 1, 'two': 2, 'three': 3}

>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

>>> d = dict([('two', 2), ('one', 1), ('three', 3)])

>>> e = dict({'three': 3, 'one': 1, 'two': 2})

>>> a == b == c == d == e

True
```

Si tenemos un diccionario vacío, al ser un objeto mutable, también podemos construir el diccionario de la siguiente manera.

```
>>> dict1 = {}

>>> dict1["one"]=1

>>> dict1["two"]=2

>>> dict1["three"]=3
```

Operaciones básicas con diccionarios

```
>>> a = dict(one=1, two=2, three=3)
```

len(): Devuelve número de elementos del diccionario.

```
>>> len(a)

3
```

Indexación: Podemos obtener el valor de un campo o cambiarlo (si no existe el campo nos da una excepción KeyError):

```
>>> a["one"]
1
>>> a["one"]+=1
>>> a
{'three': 3, 'one': 2, 'two': 2}
```

del():Podemos eliminar un elemento, si no existe el campo nos da una excepción KeyError:

```
>>> del(a["one"])
>>> a
{'three': 3, 'two': 2}
```

Operadores de pertenencia: key in d y key not in d.

```
>>> "two" in a
True
```

iter(): Nos devuelve un iterador de las claves.

```
>>> next(iter(a))
'three'
```

Los diccionarios son tipos mutables

Los diccionarios, al igual que las listas, son tipos de datos mutable. Por lo tanto podemos encontrar situaciones similares a las que explicamos en su momentos con las listas.

```
>>> a = dict(one=1, two=2, three=3)
```

```
>>> a["one"]=2
>>> del(a["three"])
>>> a
{'one': 2, 'two': 2}

>>> a = dict(one=1, two=2, three=3)
>>> b = a
>>> del(a["one"])
>>> b
{'three': 3, 'two': 2}
```

En este caso para copiar diccionarios vamos a usar el método copy():

```
>>> a = dict(one=1, two=2, three=3)
>>> b = a.copy()
>>> a["one"]=1000
>>> b
{'three': 3, 'one': 1, 'two': 2}
```

Métodos principales de diccionarios

dict1.clear dict1.get dict1.pop dict1.update

dict1.copy dict1.items dict1.popitem dict1.values

dict1.fromkeys dict1.keys dict1.setdefault

Métodos de eliminación: clear

```
>>> dict1 = dict(one=1, two=2, three=3)
```

```
>>> dict1.clear()
>>> dict1
{}
```

Métodos de agregado y creación: copy, dict.fromkeys, update, setdefault

```
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict2 = dict1.copy()

>>> dict.fromkeys(["one","two","three"])
{'one': None, 'two': None, 'three': None}
>>> dict.fromkeys(["one","two","three"],100)
{'one': 100, 'two': 100, 'three': 100}

>>> dict1 = dict(one=1, two=2, three=3)
>>> dict2 = {'four':4,'five':5}
>>> dict1.update(dict2)
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}

>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.setdefault("four",4)
4
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> dict1.setdefault("one",-1)
```

```
1
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

Métodos de retorno: get, pop, popitem, items, keys, values

```
>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.get("one")
1
>>> dict1.get("four")
>>> dict1.get("four","no existe")
'no existe'

>>> dict1.pop("one")
1
>>> dict1
{'two': 2, 'three': 3}
>>> dict1.pop("four")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'four'
>>> dict1.pop("four","no existe")
'no existe'

>>> dict1 = dict(one=1, two=2, three=3)
>>> dict1.popitem()
```

('one', 1)

>>> dict1

{'two': 2, 'three': 3}

>>> dict1 = dict(one=1, two=2, three=3)

>>> dict1.items()

dict_items([('one', 1), ('two', 2), ('three', 3)])

>>> dict1.keys()

dict_keys(['one', 'two', 'three'])

El tipo de datos dictviews

Los tres últimos métodos devuelven un objeto de tipo dictviews.

Esto devuelve una vista dinámica del ciccionario, por ejemplo:

>>> dict1 = dict(one=1, two=2, three=3)

>>> i = dict1.items()

>>> i

dict_items([('one', 1), ('two', 2), ('three', 3)])

>>> dict1["four"]=4

>>> i

dict_items([('one', 1), ('two', 2), ('three', 3), ('four', 4)])

Es este tipo de datos podemos usar las siguientes funciones:

- len(): Devuelve número de elementos de la vista.
- iter(): Nos devuelve un iterador de las claves, valores o ambas.
- x in dictview: Devuelve True si x está en las claves o valores.

Recorrido de diccionarios

Podemos recorrer las claves:

```
>>> for clave in dict1.keys():
...     print(clave)
one
two
three
```

Podemos recorrer los valores:

```
>>> for valor in dict1.values():
...     print(valor)
1
2
3
```

O podemos recorrer ambos:

```
>>> for clave,valor in dict1.items():
...     print(clave,"->",valor)
one -> 1
two -> 2
three -> 3
```

Ejercicios de diccionarios

1. Escribe un programa que lea una cadena y devuelva un diccionario con la cantidad de apariciones de cada palabra en la cadena. Por ejemplo, si recibe "Qué lindo día que hace hoy" debe devolver: 'que': 2, 'lindo': 1, 'día': 1, 'hace': 1, 'hoy': 1

2. Tenemos guardado en un diccionario los códigos morse correspondientes a cada carácter. Escribir un programa que lea una palabra y la muestra usando el código morse.
3. Continuar el programa: ahora nos pide un código morse donde cada letra está separada por espacios y nos da la cadena correspondiente.
4. Suponga un diccionario que contiene como clave el nombre de una persona y como valor una lista con todas sus "gustos". Desarrolle un programa que agregue "gustos" a la persona:

- Si la persona no existe la agregue al diccionario con una lista que contiene un solo elemento.
- Si la persona existe y el gusto existe en su lista, no tiene ningún efecto.
- Si la persona existe y el gusto no existe en su lista, agrega el gusto a la lista.

Se deja de pedir personas cuando introducimos el carácter "*".

Lectura y escritura de ficheros de textos

Función open()

La función open() se utiliza normalmente con dos parámetros (fichero con el que vamos a trabajar y modo de acceso) y nos devuelve un objeto de tipo fichero.

```
>>> f = open("ejemplo.txt","w")

>>> type(f)

<class '_io.TextIOWrapper'>

>>> f.close()
```

Modos de acceso

Los modos que podemos indicar son los siguientes:

Añadido en modo binario. Crea si éste no existe

Modo	Comportamiento	Puntero
r	Solo lectura	Al inicio del archivo
rb	Solo lectura en modo binario	
r+	Lectura y escritura	Al inicio del archivo
rb+	Lectura y escritura binario	Al inicio del archivo
w	Solo escritura. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
wb	Solo escritura en modo binario. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
w+	Escritura y lectura. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
wb+	Escritura y lectura binaria. Sobreescribe si existe. Crea el archivo si no existe.	Al inicio del archivo
a	Añadido (agregar contenido). Crea el archivo si no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al

		comienzo.
ab		Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
a+	Añadido y lectura. Crea el archivo si no existe.	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.
ab+	Añadido y lectura en binario. Crea el archivo si no existe	Si el archivo existe, al final de éste. Si el archivo no existe, al comienzo.

Como podemos comprobar podemos trabajar con ficheros binarios y con ficheros de textos.

Codificación de caracteres

Si trabajamos con fichero de textos podemos indicar también el parámetro encoding que será la codificación de caracteres utilizadas al trabajar con el fichero, por defecto se usa la indicada en el sistema:

```
>>> import locale

>>> locale.getpreferredencoding()

'UTF-8'
```

Y por último también podemos indicar el parámetro errores que controla el comportamiento cuando se encuentra con algún error al codificar o decodificar caracteres.

Objeto fichero

Al abrir un fichero con un determinado modo de acceso con la función open() se nos devuelve un objeto fichero. El fichero abierto siempre hay que cerrarlo con el método close():

```
>>> f = open("ejemplo.txt","w")

>>> type(f)

<class '_io.TextIOWrapper'>

>>> f.close()
```

Se pueden acceder a las siguientes propiedades del objeto file:

- closed: retorna True si el archivo se ha cerrado. De lo contrario, False.
- mode: retorna el modo de apertura.

- name: retorna el nombre del archivo
- encoding: retorna la codificación de caracteres de un archivo de texto

Podemos abrirlo y cerrarlo en la misma instrucción con la siguiente estructura:

```
>>> with open("ejemplo.txt", "r") as archivo:
...    contenido = archivo.read()
>>> archivo.closed
True
```

Métodos principales

Métodos de lectura

```
>>> f = open("ejemplo.txt","r")
>>> f.read()
'Hola que tal\n'

>>> f = open("ejemplo.txt","r")
>>> f.read(4)
'Hola'
>>> f.read(4)
' que'
>>> f.tell()
8
>>> f.seek(0)
>>> f.read()
'Hola que tal\n'

>>> f = open("ejemplo2.txt","r")
>>> f.readline()
'Línea 1\n'
```

```
>>> f.readline()
'Línea 2\n'
>>> f.seek(0)
0
>>> f.readlines()
['Línea 1\n', 'Línea 2\n']
```

Métodos de escritura

```
>>> f = open("ejemplo3.txt","w")
>>> f.write("Prueba 1\n")
9
>>> print("Prueba 2\n",file=f)
>>> f.writelines(["Prueba 3","Prueba 4"])
>>> f.close()
>>> f = open("ejemplo3.txt","r")
>>> f.read()
'Prueba 1\nPrueba 2\n\nPrueba 3Prueba 4'
```

Recorrido de ficheros

```
>>> with open("ejemplo3.txt","r") as fichero:
...    for linea in fichero:
...        print(linea)
```

Gestionar ficheros CSV

Módulo CSV

El módulo cvs nos permite trabajar con ficheros CSV. Un fichero CSV (comma-separated values) es un tipo de documento en formato abierto sencillo para representar datos en

forma de tabla, en las que las columnas se separan por comas (o por otro carácter).

Leer ficheros CSV

Para leer un fichero CSV utilizamos la función reader():

```
>>> import csv

>>> fichero = open("ejemplo1.csv")

>>> contenido = csv.reader(fichero)

>>> list(contenido)

[['4/5/2015 13:34', 'Apples', '73'], ['4/5/2015 3:41', 'Cherries', '85'], ['4/6/2015 12:46', 'Pears', '14'], ['4/8/2015 8:59', 'Oranges', '52'], ['4/10/2015 2:07', 'Apples', '152'], ['4/10/2015 18:10', 'Bananas', '23'], ['4/10/2015 2:40', 'Strawberries', '98']]

>>> list(contenido)

[]

>>> fichero.close()
```

Podemos guardar la lista obtenida en una variable y acceder a ella indicando fila y columna.

```
...

>>> datos = list(contenido)

>>> datos[0][0]

'4/5/2015 13:34'

>>> datos[1][1]

'Cherries'

>>> datos[2][2]

'14'
```

Por supuesto podemos recorrer el resultado:

```
...

>>> for row in contenido:

        print("Fila "+str(contenido.line_num)+" "+str(row))
```

Fila 1 ['4/5/2015 13:34', 'Apples', '73']

Fila 2 ['4/5/2015 3:41', 'Cherries', '85']

Fila 3 ['4/6/2015 12:46', 'Pears', '14']

Fila 4 ['4/8/2015 8:59', 'Oranges', '52']

Fila 5 ['4/10/2015 2:07', 'Apples', '152']

Fila 6 ['4/10/2015 18:10', 'Bananas', '23']

Fila 7 ['4/10/2015 2:40', 'Strawberries', '98']

Veamos otro ejemplo un poco más complejo:

```
>>> import csv
>>> fichero = open("ejemplo2.csv")
>>> contenido = csv.reader(fichero,quotechar='"')
>>> for row in contenido:
...   print(row)
...
['Año', 'Marca', 'Modelo', 'Descripción', 'Precio']
['1997', 'Ford', 'E350', 'ac, abs, moon', '3000.00']
['1999', 'Chevy', 'Venture "Extended Edition"', '', '4900.00']
['1999', 'Chevy', 'Venture "Extended Edition, Very Large"', '', '5000.00']
['1996', 'Jeep', 'Grand Cherokee', 'MUST SELL!\nair, moon roof, loaded', '4799.00']
```

Escribir ficheros CSV

```
>>> import csv
>>> fichero = open("ejemplo3.csv","w")
>>> contenido = csv.writer(fichero)
>>> contenido.writerow(['4/5/2015 13:34', 'Apples', '73'])
>>> contenido.writerows(['4/5/2015 3:41', 'Cherries', '85'],['4/6/2015 12:46', 'Pears', '14'])
>>> fichero.close()
```

$ cat ejemplo3.csv

4/5/2015 13:34,Apples,73

4/5/2015 3:41,Cherries,85

4/6/2015 12:46,Pears,14

Gestionar ficheros json

El módulo json nos permite gestionar ficheros con formato JSON (JavaScript Object Notation).

La correspondencia entre JSON y Python la podemos resumir en la siguiente tabla:

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Leer ficheros json

Desde una cadena de caracteres:

```
>>> import json
>>> datos_json='{"nombre":"carlos","edad":23}'
>>> datos = json.loads(datos_json)
>>> type(datos)
```

<class 'dict'>

>>> print(datos)

{'nombre': 'carlos', 'edad': 23}

Desde un fichero:

>>> with open("ejemplo1.json") as fichero:

... datos=json.load(fichero)

>>> type(datos)

<class 'dict'>

>>> datos

{'bookstore': {'book': [{'_category': 'COOKING', 'price': '30.00', 'author': 'Giada De Laurentiis', 'title': {'__text': 'Everyday Italian', '_lang': 'en'}, 'year': '2005'}, {'_category': 'CHILDREN', 'price': '29.99', 'author': 'J K. Rowling', 'title': {'__text': 'Harry Potter', '_lang': 'en'}, 'year': '2005'}, {'_category': 'WEB', 'price': '49.99', 'author': ['James McGovern', 'Per Bothner', 'Kurt Cagle', 'James Linn', 'Vaidyanathan Nagarajan'], 'title': {'__text': 'XQuery Kick Start', '_lang': 'en'}, 'year': '2003'}, {'_category': 'WEB', 'price': '39.95', 'author': 'Erik T. Ray', 'title': {'__text': 'Learning XML', '_lang': 'en'}, 'year': '2003'}]}}

Escribir ficheros json

>>> datos = {'isCat': True, 'miceCaught': 0, 'name': 'Zophie','felineIQ': None}

>>> fichero = open("ejemplo2.json","w")

>>> json.dump(datos,fichero)

>>> fichero.close()

cat ejemplo2.json

{"miceCaught": 0, "name": "Zophie", "felineIQ": null, "isCat": true}

Excepciones

Errores sintácticos y errores de ejecución

Veamos un ejemplo de error sintáctico:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

Una excepción o un error de ejecución se produce durante la ejecución del programa. Las excepciones se pueden manejar para que no termine el programa. Veamos algunos ejemplos de excepciones:

```
>>> 4/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> a+4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

```
>>> "2"+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Hemos obtenido varias excepciones: ZeroDivisionError, NameError y TypeError. Puedes ver la lista de excepciones y su significado.

Manejando excepciones. try, except, else, finally

Veamos un ejemplo simple cómo podemos tratar una excepción:

```
>>> while True:
...   try:
...     x = int(input("Introduce un número:"))
...     break
...   except ValueError:
...     print ("Debes introducir un número")
```

1. Se ejecuta el bloque de instrucciones de try.
2. Si no se produce la excepción, el bloque de excepción no se ejecuta y continúa la ejecución secuencia.
3. Si se produce una excepción, el resto del bloque try no se ejecuta, si la excepción que se ha produce corresponde con la indicada en excepción se salta a ejecutar el bloque de instrucciones except.
4. Si la excepción producida no se corresponde con las indicadas en excepción se pasa a otra instrucción try, si finalmente no hay un manejador nos dará un error y el programa terminará.

Un bloque except puede manejar varios tipos de excepciones:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Si quiero controlar varios tipos de excepciones puedo poner varios bloques except. Teniendo en cuenta que en el último, si quiero no indico el tipo de excepción:

```
>>> try:
...   print (10/int(cad))
... except ValueError:
...   print("No se puede convertir a entero")
... except ZeroDivisionError:
...   print("No se puede dividir por cero")
... except:
...   print("Otro error")
```

Se puede utilizar también la cláusula else:

```
>>> try:
```

```
...   print (10/int(cad))
... except ValueError:
...   print("No se puede convertir a entero")
... except ZeroDivisionError:
...   print("No se puede dividir por cero")
... else:
...   print("Otro error")
```

Por último indicar que podemos indicar una cláusula finally para indicar un bloque de instrucciones que siempre se debe ejecutar, independientemente de la excepción se haya producido o no.

```
>>> try:
...     result = x / y
... except ZeroDivisionError:
...     print("División por cero!")
... else:
...     print("El resultado es", result)
... finally:
...     print("Terminamos el programa")
```

Obteniendo información de las excepciones

```
>>> cad = "a"
>>> try:
...   i = int(cad)
... except ValueError as error:
...   print(type(error))
...   print(error.args)
...   print(error)
...
```

<class 'ValueError'>

("invalid literal for int() with base 10: 'a'",)

invalid literal for int() with base 10: 'a'

Propagando excepciones. raise

Si construimos una función donde se maneje una excepción podemos hacer que la excepción se envíe a la función desde la que la hemos llamado. Para ello utilizamos la instrucción raise. Veamos algunos ejemplos:

```
def dividir(x,y):
    try:
        return x/y
    except ZeroDivisionError:
        raise
```

Con raise también podemos propagar una excepción en concreto:

```
def nivel(numero):
    if numero<0:
        raise ValueError("El número debe ser positivo:"+str(numero))
    else:
        return numero
```

Módulos y paquetes

- Módulo: Cada uno de los ficheros .py que nosotros creamos se llama módulo. Los elementos creados en un módulo (funciones, clases, ...) se pueden importar para ser utilizados en otro módulo. El nombre que vamos a utilizar para importar un módulo es el nombre del fichero.

- Paquete: Para estructurar nuestros módulos podemos crear paquetes. Un paquete, es una carpeta que contiene archivos .py. Pero, para que una carpeta pueda ser considerada un paquete, debe contener un archivo de inicio llamado __init__.py. Este archivo, no necesita contener ninguna instrucción. Los paquetes, a la vez, también pueden contener otros sub-paquetes.

Ejecutando módulos como scripts

Si hemos creado un módulo, donde hemos definido dos funciones y hemos hecho un programa principal donde se utilizan dichas funciones, tenemos dos opciones: ejecutar ese módulo como un script o importar este módulo desde otro, para utilizar sus funciones. Por ejemplo, si tenemos un fichero llamado potencias.py:

```
#!/usr/bin/env python


def cuadrado(n):
    return(n**2)
def cubo(n):
    return(n**3)
if __name__ == "__main__":
    print(cuadrado(3))
    print(cubo(3))
```

En este caso, cuando lo ejecuto como un script:

```
$ python3 potencias.py
```

El nombre que tiene el módulo es __main__, por lo tanto se ejecutará el programa principal.

Además este módulo se podrá importar (como veremos en el siguiente apartado) y el programa principal no se tendrá en cuenta.

Importando módulos: import, from

Para importar un módulo completo tenemos que utilizar las instrucciones import. lo podemos importar de la siguiente manera:

```
>>> import potencias

>>> potencias.cuadrado(3)

9

>>> potencias.cubo(3)

27
```

Namespace y alias

Para acceder (desde el módulo donde se realizó la importación), a cualquier elemento del módulo importado, se realiza mediante el **namespace**, seguido de un punto (.) y el nombre del elemento que se desee obtener. En Python, un **namespace**, es el nombre que se ha indicado luego de la palabra import, es decir la ruta (namespace) del módulo.

Es posible también, abreviar los **namespaces** mediante un **alias**. Para ello, durante la importación, se asigna la palabra clave as seguida del alias con el cuál nos referiremos en el futuro a ese namespace importado:

```
>>> import potencias as p

>>> p.cuadrado(3)

9
```

Importando elementos de un módulo: from...import

Para no utilizar el **namespace** podemos indicar los elementos concretos que queremos importar de un módulo:

```
>>> from potencias import cubo

>>> cubo(3)

27
```

Podemos importar varios elementos separándolos con comas:

```
>>> from potencias import cubo,cuadrado
```

Podemos tener un problema al importar dos elementos de dos módulos que se llamen igual. En este caso tengo que utilizar **alias**:

```
>>> from potencias import cuadrado as pc

>>> from dibujos import cuadrado as dc

>>> pc(3)

9

>>> dc()

Esto es un cuadrado
```

Importando módulos desde paquetes

Si tenemos un módulo dentro de un paquete la importación se haría de forma similar. tenemos un paquete llamado operaciones:

```
$ cd operaciones

$ ls

__init.py__  potencias.py
```

Para importarlo:

```
>>> import operaciones.potencias

>>> operaciones.potencias.cubo(3)

27


>>> from operaciones.potencias import cubo

>>> cubo(3)

27
```

Función dir()

La función dir() nos permite averiguar los elementos definidos en un módulo:

```
>>> import potencias

>>> dir(potencias)

['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'cuadrado', 'cubo']
```

¿Dónde se encuentran los módulos?

Los módulos estándar (como math o sys por motivos de eficiencia están escritos en C e incorporados en el intérprete (builtins).

Para obtener la lista de módulos builtins:

```
>>> import sys

>>> sys.builtin_module_names

('_ast', '_bisect', '_codecs', '_collections', '_datetime', '_elementtree', '_functools', '_heapq', '_imp', '_io', '_locale', '_md5', '_operator', '_pickle', '_posixsubprocess', '_random', '_sha1', '_sha256', '_sha512', '_socket', '_sre', '_stat', '_string', '_struct', '_symtable', '_thread', '_tracemalloc', '_warnings', '_weakref', 'array', 'atexit', 'binascii', 'builtins', 'errno', 'faulthandler', 'fcntl', 'gc', 'grp', 'itertools', 'marshal', 'math', 'posix', 'pwd', 'pyexpat', 'select', 'signal', 'spwd', 'sys', 'syslog', 'time', 'unicodedata', 'xxsubtype', 'zipimport', 'zlib')
```

Los demás módulos que podemos importar se encuentran guardados en ficheros, que se encuentra en los path indicados en el intérprete:

```
>>> sys.path

['', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86_64-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3/dist-packages']
```

Módulos estándares: módulos de sistema

Python tiene sus propios módulos, los cuales forman parte de su librería de módulos estándar, que también pueden ser importados. En esta unidad vamos a estudiar las funciones principales de módulos relacionados con el sistema operativo.

Módulo os

El módulo nos permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que nos refieren información sobre el entorno del mismo y nos

permiten manipular la estructura de directorios.

Descripción	Método
Saber si se puede acceder a un archivo o directorio	os.access(path, modo_de_acceso)
Conocer el directorio actual	os.getcwd()
Cambiar de directorio de trabajo	os.chdir(nuevo_path)
Cambiar al directorio de trabajo raíz	os.chroot()
Cambiar los permisos de un archivo o directorio	os.chmod(path, permisos)
Cambiar el propietario de un archivo o directorio	os.chown(path, permisos)
Crear un directorio	os.mkdir(path[, modo])
Crear directorios recursivamente	os.mkdirs(path[, modo])
Eliminar un archivo	os.remove(path)
Eliminar un directorio	os.rmdir(path)
Eliminar directorios recursivamente	os.removedirs(path)
Renombrar un archivo	os.rename(actual, nuevo)
Crear un enlace simbólico	os.symlink(path, nombre_destino)

```
>>> import os
>>> os.getcwd()
'/home/jose/github/curso_python3/curso/u40'
>>> os.chdir("..")
>>> os.getcwd()
'/home/jose/github/curso_python3/curso'
```

El módulo os también nos provee del submódulo path (os.path) el cual nos permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios.

Descripción	Método
Ruta absoluta	os.path.abspath(path)
Directorio base	os.path.basename(path)
Saber si un directorio existe	os.path.exists(path)
Conocer último acceso a un directorio	os.path.getatime(path)
Conocer tamaño del directorio	os.path.getsize(path)
Saber si una ruta es absoluta	os.path.isabs(path)
Saber si una ruta es un archivo	os.path.isfile(path)
Saber si una ruta es un directorio	os.path.isdir(path)
Saber si una ruta es un enlace simbólico	os.path.islink(path)
Saber si una ruta es un punto de montaje	os.path.ismount(path)

Ejecutar comandos del sistema operativo. Módulo subprocess

Con la función system() del módulo nos permite ejecutar comandos del sistema operativo.

```
>>> os.system("ls")
curso  modelo.odp  README.md
0
```

La función nos devuelve un código para indicar si la instrucción se ha ejecutado con éxito.

Tenemos otra forma de ejecutar comandos del sistema operativo que nos da más funcionalidad, por ejemplo nos permite guardar la salida del comando en una variable. Para ello podemos usar el módulo subprocess

```
>>> import subprocess
>>> subprocess.call("ls")
curso  modelo.odp  README.md
0
```

```
>>> salida=subprocess.check_output("ls")
>>> print(salida.decode())
curso
modelo.odp
README.md

>>> salida=subprocess.check_output(["df","-h"])

>>> salida = subprocess.Popen(["df","-h"], stdout=subprocess.PIPE)
>>> salida.communicate()[0]
```

Módulo shutil

El módulo shutil de funciones para realizar operaciones de alto nivel con archivos y directorios. Dentro de las operaciones que se pueden realizar está copiar, mover y borrar archivos y directorios; y copiar los permisos y el estado de los archivos.

Descripción	**Método**
Copia un fichero completo o parte	shutil.copyfileobj(fsrc, fdst[, length])
Copia el contenido completo (sin metadatos) de un archivo	shutil.copyfile(src, dst, *, follow_symlinks=True)
copia los permisos de un archivo origen a uno destino	shutil.copymode(src, dst, *, follow_symlinks=True)
Copia los permisos, la fecha-hora del último acceso, la fecha-hora de la última modificación y los atributos de un archivo origen a un archivo destino	shutil.copystat(src, dst, *, follow_symlinks=True)
Copia un archivo (sólo datos y permisos)	shutil.copy(src, dst, *, follow_symlinks=True)
Copia archivos (datos, permisos y metadatos)	shutil.move(src, dst, copy_function=copy2)
Obtiene información del espacio total, usado y libre, en	shutil.disk_usage(path)

bytes	
Obtener la ruta de un archivo ejecutable	shutil.chown(path, user=None, group=None)
Saber si una ruta es un enlace simbólico	shutil.which(cmd, mode=os.F_OK | os.X_OK, path=None)

Módulos sys

El módulo sys es el encargado de proveer variables y funcionalidades, directamente relacionadas con el intérprete.

Algunas variables definidas en el módulo:

Variable	Descripción
sys.argv	Retorna una lista con todos los argumentos pasados por línea de comandos. Al ejecutar python modulo.py arg1 arg2, retornará una lista: ['modulo.py', 'arg1', 'arg2']
sys.executable	Retorna el path absoluto del binario ejecutable del intérprete de Python
sys.platform	Retorna la plataforma sobre la cuál se está ejecutando el intérprete
sys.version	Retorna el número de versión de Python con información adicional

Y algunos métodos:

Método	Descripción
sys.exit()	Forzar la salida del intérprete
sys.getdefaultencoding()	Retorna la codificación de caracteres por defecto

Ejecución de scripts con argumentos

Podemos enviar información (argumentos) a un programa cuando se ejecuta como un script, por ejemplo:

```
#!/usr/bin/env python
import sys
print("Has introducido",len(sys.argv),"argumento")
suma=0
for i in range(1,len(sys.argv)):
        suma=suma+int(sys.argv[i])
print("La suma es ",suma)
```

```
$  python3 sumar.py 3 4 5
Has introducido 4 argumento
La suma es  12
```

Módulos estándares: módulos matemáticos

Módulo math

El módulo math nos proporciona distintas funciones y operaciones matemáticas.

```
>>> import math
>>> math.factorial(5)
120
>>> math.pow(2,3)
8.0
>>> math.sqrt(7)
2.6457513110645907
>>> math.cos(1)
0.5403023058681398
>>> math.pi
3.141592653589793
```

```
>>> math.log(10)
2.302585092994046
```

Módulo fractions

El módulo fractions nos permite trabajar con fracciones.

```
>>> from fractions import Fraction
>>> a=Fraction(2,3)
>>> b=Fraction(1.5)
>>> b
Fraction(3, 2)
>>> c=a+b
>>> c
Fraction(13, 6)
```

Módulo statistics

El módulo statistics nos proporciona funciones para hacer operaciones estadísticas.

```
>>> import statistics
>>> statistics.mean([1,4,5,2,6])
3.6

>>> statistics.median([1,4,5,2,6])
4
```

Módulo random

El módulo random nos permite generar datos pseudoaleatorios.

```
>>> import random
>>> random.randint(10,100)
```

```
>>> random.choice(["hola","que","tal"])
'que'

>>> frutas=["manzanas","platanos","naranjas"]
>>> random.shuffle(frutas)
>>> frutas
['naranjas', 'manzanas', 'platanos']

>>> lista = [1,3,5,2,7,4,9]
>>> numeros=random.sample(lista,3)
>>> numeros
[1, 2, 4]
```

Módulos estándares: módulos de hora y fechas

Módulo time

El tiempo es medido como un número real que representa los segundos transcurridos desde el 1 de enero de 1970. Por lo tanto es imposible representar fechas anteriores a esta y fechas a partir de 2038 (tamaño del float en la librería C (32 bits)).

```
>>> import time
>>> time.time()
1488619835.7858684
```

Para convertir la cantidad de segundos a la fecha y hora local:

```
>>> tiempo = time.time()
>>> time.localtime(tiempo)
```

time.struct_time(tm_year=2017, tm_mon=3, tm_mday=4, tm_hour=10, tm_min=37, tm_sec=19, tm_wday=5, tm_yday=63, tm_isdst=0)

Si queremos obtener la fecha y hora actual:

```
>>> time.localtime()
```

time.struct_time(tm_year=2017, tm_mon=3, tm_mday=4, tm_hour=10, tm_min=37, tm_sec=30, tm_wday=5, tm_yday=63, tm_isdst=0)

Nos devuelve a una estructura a la que podemos acceder a sus distintos campos.

```
>>> tiempo = time.localtime()
>>> tiempo.tm_year
```

2017

Podemos representar la fecha y hora como una cadena:

```
>>> time.asctime()
```

'Sat Mar 4 10:41:41 2017'

```
>>> time.asctime(tiempo)
```

'Sat Mar 4 10:39:21 2017'

O con un determinado formato:

```
>>> time.strftime('%d/%m/%Y %H:%M:%S')
```

'04/03/2017 10:44:52'

```
>>> time.strftime('%d/%m/%Y %H:%M:%S',tiempo)
```

'04/03/2017 10:39:21'

Módulo datetime

Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de tiempo.

```
>>> from datetime import datetime
```

```
>>> datetime.now()
datetime.datetime(2017, 3, 4, 10, 52, 12, 859564)
>>> datetime.now().day,datetime.now().month,datetime.now().year
(4, 3, 2017)
```

Para comparar fechas y horas:

```
>>> from datetime import datetime, date, time, timedelta
>>> hora1 = time(10,5,0)
>>> hora2 = time(23,15,0)
>>> hora1>hora2
False

>>> fecha1=date.today()
>>> fecha2=fecha1+timedelta(days=2)
>>> fecha1
datetime.date(2017, 3, 4)
>>> fecha2
datetime.date(2017, 3, 6)
>>> fecha1<fecha2
True
```

Podemos imprimir aplicando un formato:

```
>>> fecha1.strftime("%d/%m/%Y")
'04/03/2017'
>>> hora1.strftime("%H:%M:%S")
'10:05:00'
```

Podemos convertir una cadena a un datetime:

```
>>> tiempo = datetime.strptime("12/10/2017","%d/%m/%Y")
```

Y podemos trabajar con cantidades (segundos, minutos, horas, días, semanas,...) con timedelta:

```
>>> hoy = date.today()

>>> ayer = hoy - timedelta(days=1)

>>> diferencia=hoy -ayer

>>> diferencia

datetime.timedelta(1)
```

```
>>> fecha1=datetime.now()

>>> fecha2=datetime(1995,10,12,12,23,33)

>>> diferencia=fecha1-fecha2

>>> diferencia

datetime.timedelta(7813, 81981, 333199)
```

Módulo calendar

Podemos obtener el calendario del mes actual:

```
>>> año = date.today().year

>>> mes = date.today().month

>>> calendario_mes = calendar.month(año, mes)

>>> print(calendario_mes)

    March 2017

Mo Tu We Th Fr Sa Su

       1  2  3  4  5

 6  7  8  9 10 11 12

13 14 15 16 17 18 19

20 21 22 23 24 25 26
```

27 28 29 30 31

Y para mostrar todos los meses del año:

>>> print(calendar.TextCalendar(calendar.MONDAY).formatyear(2017,2, 1, 1, 2))

Instalación de módulos

Python posee una activa comunidad de desarrolladores y usuarios que desarrollan tanto los módulos estándar de python, como módulos y paquetes desarrollados por terceros.

PyPI y pip

- El *Python Package Index* o *PyPI*, es el repositorio de paquetes de software oficial para aplicaciones de terceros en el lenguaje de programación Python.
- pip: Sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python que se encuentran alojados en el repositorio *PyPI*.

Instalación de módulos python

Para instalar un nuevo paquete python tengo varias alternativas:

Utilizar el que esté empaquetado en la distribución que estés usando. Podemos tener problemas si necesitamos una versión determinada.

```
# apt-cache show python3-requests
...
Version: 2.4.3-6
...
```

1.

Instalar pip en nuestro equipo, y como superusuario instalar el paquete python que nos interesa. Esta solución nos puede dar muchos problemas, ya que podemos romper las dependencias entre las versiones de nuestros paquetes python instalados en el sistema y algún paquete puede dejar de funcionar.

```
# pip search requests
...
```

requests (2.13.0) - Python HTTP for Humans.

...

2.

3. Utilizar entornos virtuales: es un mecanismo que me permite gestionar programas y paquetes python sin tener permisos de administración, es decir, cualquier usuario sin privilegios puede tener uno o más "espacios aislados" (ya veremos más adelante que los entornos virtuales se guardan en directorios) donde poder instalar distintas versiones de programas y paquetes python. Para crear los entornos virtuales vamos a usar el programa virtualenv o el módulo venv.

Creando entornos virtuales con virtualenv

Podemos utilizar este software para trabajar con cualquier distribución de python, pero evidentemente es obligatorio si estamos trabajando con python 2.x o python 3.x (una versión anterior a la 3.3).

apt-get install python-virtualenv

Si queremos crear un entorno virtual con python3:

$ virtualenv -p /usr/bin/python3 entorno2

La opción -p nos permite indicar el intérprete que se va a utilizar en el entorno.

Para activar nuestro entorno virtual:

$ source entorno2/bin/activate

(entorno2)$

Y para desactivarlo:

(entorno2)$ deactivate

$

Creando entornos virtuales con venv

A partir de la versión 3.3 de python podemos utilizar el módulo venv para crear el entorno virtual.

Instalamos el siguiente paquete para instalar el módulos:

apt-get install python3-venv

Ahora ya como un usuario sin privilegio podemos crear un entorno virtual con python3:

$ python3 -m venv entorno3

La opción -m del intérprete nos permite ejecutar un módulo como si fuera un programa.

Para activar y desactivar el entono virtual:

$ source entorno3/bin/activate

(entorno3)$ deactivate

$

Instalando paquetes en nuestro entorno virtual

Independientemente del sistema utilizado para crear nuestro entorno virtual, una vez que lo tenemos activado podemos instalar paquetes python en él utilizando la herramienta pip (que la tenemos instalada automáticamente en nuestro entorno). Partiendo de un entorno activado, podemos, por ejemplo, instalar la última versión de django:

(entorno3)$ pip install django

Si queremos ver los paquetes que tenemos instalados con sus dependencias:

(entorno3)$ pip list

Django (1.10.5)

pip (1.5.6)

setuptools (5.5.1)

Si necesitamos buscar un paquete podemos utilizar la siguiente opción:

(entorno3)$ pip search requests

Si a continuación necesitamos instalar una versión determinada del paquete, que no sea la última, podemos hacerlo de la siguiente manera:

(entorno3)$ pip install requests=="2.12"

Si necesitamos borrar un paquete podemos ejecutar:

(entorno3)$ pip uninstall requests

Y, por supuesto para instalar la última versión, simplemente:

(entorno3)$ pip install requests

Para terminar de repasar la herramienta pip, vamos a explicar cómo podemos guardar en un fichero (que se suele llamar requirements.txt) la lista de paquetes instalados, que nos permite de manera sencilla crear otro entorno virtual en otra máquina con los mismos paquetes instalados. Para ello vamos a usar la siguiente opción de pip:

(entorno3)$ pip freeze

Django==1.10.5

requests==2.13.0

Y si queremos guardar esta información en un fichero que podamos distribuir:

(entorno3)$ pip freeze > requirements.txt

De tal manera que otro usuario, en otro entorno, teniendo este fichero puede reproducirlo e instalar los mismos paquetes de la siguiente manera:

(entorno4)$ pip install -r requirements.txt

Introducción a las funciones

Introducción a la programación estructurada y modular

La programación estructurada es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de ordenador, utilizando únicamente subrutinas (funciones o procedimientos) y tres estructuras: secuencia, alternativas y repetitivas.

La programación modular es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación (divide y vencerás).

La programación estructurada y modular se lleva a cabo en python3 con la definición de funciones.

Definición de funciones

Veamos un ejemplo de definición de función:

```
>>> def factorial(n):
...     """Calcula el factorial de un número"""
...     resultado = 1
...     for i in range(1,n+1):
...         resultado*=i
...     return resultado
```

Podemos obtener información de la función:

```
>>> help(factorial)
Help on function factorial in module __main__:

factorial(n)
    Calcula el factorial de un número
```

Y para utilizar la función:

```
>>> factorial(5)

120
```

Ámbito de variables. Sentencia global

Una variable local se declara en su ámbito de uso (en el programa principal y dentro de una función) y una global fuera de su ámbito para que se pueda utilizar en cualquier función que la declare como global.

```
>>> def operar(a,b):

...     global suma

...     suma = a + b

...     resta = a - b

...     print(suma,resta)

...

>>> operar(4,5)

9 -1

>>> resta

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

NameError: name 'resta' is not defined

>>> suma

9
```

Podemos definir variables globales, que serán visibles en todo el módulo. Se recomienda declararlas en mayúsculas:

```
>>> PI = 3.1415

>>> def area(radio):

...     return PI*radio**2

...

>>> area(2)
```

12.566

Parámetros formales y reales

- Parámetros formales: Son las variables que recibe la función, se crean al definir la función. Su contenido lo recibe al realizar la llamada a la función de los parámetros reales. Los parámetros formales son variables locales dentro de la función.

- Parámetros reales: Son expresiones que se utilizan en la llamada de la función, sus valores se copian en los parámetros formales.

Paso de parámetro por valor o por referencia

En Python el paso de parámetros es siempre por referencia. El lenguaje no trabaja con el concepto de variables sino con objetos y referencias. Al realizar la asignación a = 1 no se dice que "a contiene el valor 1" sino que "a referencia a 1". Así, en comparación con otros lenguajes, podría decirse que en Python los parámetros siempre se pasan por referencia.

Evidentemente si se pasa un valor de un objeto inmutable, su valor no se podrá cambiar dentro de la función:

```
>>> def f(a):
...     a=5
>>> a=1
>>> f(a)
>>> a
1
```

Sin embargo si pasamos un objeto de un tipo mutable, si podremos cambiar su valor:

```
>>> def f(lista):
...   lista.append(5)
...
>>> l = [1,2]
>>> f(l)
>>> l
[1, 2, 5]
```

Aunque podemos cambiar el parámetro real cuando los objetos pasados son de tipo mutables, no es recomendable hacerlo en Python. En otros lenguajes es necesario porque no tenemos opción de devolver múltiples valores, pero como veremos en Python podemos devolver tuplas o lista con la instrucción return.

Llamadas a una función

Cuando se llama a una función se tienen que indicar los parámetros reales que se van a pasar. La llamada a una función se puede considerar una expresión cuyo valor y tipo es el retornado por la función. Si la función no tiene una instrucción return el tipo de la llamada será None.

```
>>> def cuadrado(n):
...   return n*n


>>> a=cuadrado(2)
>>> cuadrado(3)+1
10
>>> cuadrado(cuadrado(4))
256
>>> type(cuadrado(2))
<class 'int'>
```

Cuando estamos definiendo una función estamos creando un objeto de tipo function.

```
>>> type(cuadrado)
<class 'function'>
```

Y por lo tanto puedo guardar el objeto función en otra variable:

```
>>> c=cuadrado
>>> c(4)
16
```

Conceptos avanzados sobre funciones

Tipos de argumentos: posicionales o keyword

Tenemos dos tipos de parámetros: los posicionales donde el parámetro real debe coincidir en posición con el parámetro formal:

```
>>> def sumar(n1,n2):
...   return n1+n2
...
>>> sumar(5,7)
12
>>> sumar(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sumar() missing 1 required positional argument: 'n2'
```

Además podemos tener parámetros con valores por defecto:

```
>>> def operar(n1,n2,operador='+',respuesta='El resultado es '):
...   if operador=="+":
...     return respuesta+str(n1+n2)
...   elif operador=="-":
...     return respuesta+str(n1-n2)
...   else:
...     return "Error"
...
>>> operar(5,7)
'El resultado es 12'
>>> operar(5,7,"-")
'El resultado es -2'
>>> operar(5,7,"-","La resta es ")
```

'La resta es -2'

Los parámetros keyword son aquellos donde se indican el nombre del parámetro formal y su valor, por lo tanto no es necesario que tengan la misma posición. Al definir una función o al llamarla, hay que indicar primero los argumentos posicionales y a continuación los argumentos con valor por defecto (keyword).

```
>>> operar(5,7)          # dos parámetros posicionales

>>> operar(n1=4,n2=6)         # dos parámetros keyword

>>> operar(4,6,respuesta="La suma es")      # dos parámetros posicionales y uno keyword

>>> operar(4,6,respuesta="La resta es",operador="-")        # dos parámetros posicionales y dos keyword
```

Parámetro *

Un parámetro * entre los parámetros formales de una función, nos obliga a indicar los parámetros reales posteriores como keyword:

```
>>> def sumar(n1,n2,*,op="+"):

...   if op=="+":

...     return n1+n2

...   elif op=="-":

...     return n1-n2

...   else:

...     return "error"

...

>>> sumar(2,3)

5

>>> sumar(2,3,"-")

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: sumar() takes 2 positional arguments but 3 were given

>>> sumar(2,3,op="-")
```

-1

Argumentos arbitrarios (*args y **kwargs)

Para indicar un número indefinido de argumentos posicionales al definir una función, utilizamos el símbolo *:

```
>>> def sumar(n,*args):
...   resultado=n
...   for i in args:
...     resultado+=i
...   return resultado
...
>>> sumar(2)
2
>>> sumar(2,3,4)
9
```

Para indicar un número indefinido de argumentos keyword al definir una función, utilizamos el símbolo **:

```
>>> def saludar(nombre="pepe",**kwargs):
...   cadena=nombre
...   for valor in kwargs.values():
...     cadena=cadena+" "+valor
...   return "Hola "+cadena
...
>>> saludar()
'Hola pepe'
>>> saludar("juan")
'Hola juan'
>>> saludar(nombre="juan",nombre2="pepe")
```

'Hola juan pepe'

>>> saludar(nombre="juan",nombre2="pepe",nombre3="maria")

'Hola juan maria pepe'

Por lo tanto podríamos tener definiciones de funciones del tipo:

>>> def f()

>>> def f(a,b=1)

>>> def f(a,*args,b=1)

>>> def f(*args,b=1)

>>> def f(*args,b=1,*kwargs)

>>> def f(*args,*kwargs)

>>> def f(*args)

>>> def f(*kwargs)

Desempaquetar argumentos: pasar listas y diccionarios

En caso contrario es cuando tenemos que pasar parámetros que tenemos guardados en una lista o en un diccionario.

Para pasar listas utilizamos el símbolo *:

>>> lista=[1,2,3]

>>> sumar(*lista)

6

>>> sumar(2,*lista)

8

>>> sumar(2,3,*lista)

11

Podemos tener parámetros keyword guardados en un diccionario, para enviar un diccionario utilizamos el símbolo **:

```
>>> datos={"nombre":"jose","nombre2":"pepe","nombre3":"maria"}

>>> saludar(**datos)

'Hola jose maria pepe'
```

Devolver múltiples resultados

La instrucción return puede devolver cualquier tipo de resultados, por lo tanto es fácil devolver múltiples datos guardados en una lista o en un diccionario. Veamos un ejemplo en que devolvemos los datos en una tupla:

```
>>> def operar(n1,n2):

...   return (n1+n2,n1-n2,n1*n2)


>>> suma,resta,producto = operar(5,2)

>>> suma

7

>>> resta

3

>>> producto

10
```

Tipos especiales de funciones

Funciones recursivas

Una función recursiva es aquella que al ejecutarse hace llamadas a ella misma. Por lo tanto tenemos que tener "un caso base" que hace terminar el bucle de llamadas. Veamos un ejemplo:

```
>>> def factorial(numero):

...     if(numero == 0 or numero == 1):

...         return 1

...     else:
```

```
...         return numero * factorial(numero-1)

...

>>> factorial(5)

120
```

Funciones lambda

Las funciones lambda nos sirven para crear pequeñas funciones anónimas, de una sola línea sobre la marcha.

```
>>> cuadrado = lambda x: x**2

>>> cuadrado(2)
```

Como podemos notar las funciones lambda no tienen nombre. Pero gracias a que lambda crea una referencia a un objeto función, la podemos llamar.

```
>>> lambda x: x**2

<function <lambda> at 0xb74469cc>

>>>

>>> (lambda x: x**2)(3)

9
```

Otro ejemplo:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

>>> pairs.sort(key=lambda pair: pair[1])

>>> pairs

[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Decoradores

Los decoradores son funciones que reciben como parámetros otras funciones y retornan como resultado otras funciones con el objetivo de alterar el funcionamiento original de la función que se pasa como parámetro. Hay funciones que tienen en común muchas funcionalidades, por ejemplo las de manejo de errores de conexión de recursos I/O (que se

deben programar siempre que usemos estos recursos) o las de validación de permisos en las respuestas de peticiones de servidores, en vez de repetir el código de rutinas podemos abstraer, bien sea el manejo de error o la respuesta de peticiones, en una función decorador.

```
>>> def tablas(funcion):
...     def envoltura(tabla=1):
...         print('Tabla del %i:' %tabla)
...         print('-' * 15)
...         for numero in range(0, 11):
...             funcion(numero, tabla)
...         print('-' * 15)
...     return envoltura
...
>>> @tablas
... def suma(numero, tabla=1):
...     print('%2i + %2i = %3i' %(tabla, numero, tabla+numero))
...
>>> @tablas
... def multiplicar(numero, tabla=1):
...     print('%2i X %2i = %3i' %(tabla, numero, tabla*numero))


# Muestra la tabla de sumar del 1
suma()
# Muestra la tabla de sumar del 4
suma(4)
# Muestra la tabla de multiplicar del 1
multiplicar()
# Muestra la tabla de multiplicar del 10
multiplicar(10)
```

Funciones generadoras

Un generador es un tipo concreto de iterador. Es una función que permite obtener sus resultados paso a paso.

```
>>> def par(inicio,fin):
...   for i in range(inicio,fin):
...     if i % 2==0:
...       yield i


>>> datos = par(1,5)
>>> next(datos)
2
>>> next(datos)
4


>>> for i in par(20,30):
...   print(i,end=" ")
20 22 24 26 28


>>> lista_pares = list(par(1,10))
>>> lista_pares
[2, 4, 6, 8]
```

Ejercicios con funciones

1. Escribir dos funciones que permitan calcular:

- La cantidad de segundos en un tiempo dado en horas, minutos y segundos.
- La cantidad de horas, minutos y segundos de un tiempo dado en segundos.

2. Realiza una función que dependiendo de los parámetros que reciba: convierte a segundos o a horas:

- Si recibe un argumento, supone que son segundos y convierte a horas, minutos y segundos.
- Si recibe 3 argumentos, supone que son hora, minutos y segundos y los convierte a segundos.

3. Queremos hacer una función que añade a una lista los contactos de una agenda. Los contactos se van a guardar en un diccionario, y al menos debe tener el campo de nombre, el campo del teléfono, aunque puede tener más campos. Los datos se irán pidiendo por teclado, se pedirá de antemanos cuántos contactos se van a guardar. Si vamos a guardar más información en el contacto, se irán pidiendo introduciendo campos hasta que introducimos el "*".

4. Amplía el programa anterior para hacer una función de búsqueda, que reciba un conjunto de parámetros keyword y devuelve los contactos (en una lista) que coincidan con los criterios de búsqueda.

5. Realizar una función recursiva que reciba una lista y que calcule el producto de los elementos de la lista:

Programación orientada a objetos

Introducción a la Programación Orientada a Objetos

La Programación Orientado a Objetos (POO) se basa en la agrupación de objetos de distintas clases que interactúan entre sí y que, en conjunto, consiguen que un programa cumpla su propósito. En Python cualquier elemento del lenguaje pertenece a una clase y todas las clases tienen el mismo rango y se utilizan del mismo modo.

Definición de clase, objeto, atributos y métodos

- Llamamos clase a la representación abstracta de un concepto. Por ejemplo, "perro", "número entero" o "servidor web".
- Las clases se componen de atributos y métodos.
- Un objeto es cada una de las instancias de una clase.
- Los atributos definen las características propias del objeto y modifican su estado. Son datos asociados a las clases y a los objetos creados a partir de ellas.
- Un atributo de clase es compartida por todas las instancias de una clase. Se definen dentro de la clase (después del encabezado de la clase) pero nunca dentro de un método. Este tipo de variables no se utilizan con tanta frecuencia como las variables de instancia.
- Un atributo de objeto se define dentro de un método y pertenece a un objeto determinado de la clase instanciada.
- Los métodos son bloques de código (o funciones) de una clase que se utilizan para definir el comportamiento de los objetos.

Definimos nuestra primera clase:

```
>>> class clase():
...    at_clase=1
...    def metodo(self):
...      self.at_objeto=1
...
>>> type(clase)
<class 'type'>
>>> clase.at_clase
1
>>> objeto=clase()
>>> objeto.at_clase
1
>>> objeto.at_objeto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'clase' object has no attribute 'at_objeto'
>>> objeto.metodo()
>>> objeto.at_objeto
1
```

Atributos de objetos

Para definir atributos de objetos, basta con definir una variable dentro de los métodos, es una buena idea definir todos los atributos de nuestras instancias en el constructor, de modo que se creen con algún valor válido.

Método constructor init

Como hemos visto anteriormente los atributos de objetos no se crean hasta que no hemos ejecutado el método. Tenemos un método especial, llamado constructor __init__, que nos permite inicializar los atributos de objetos. Este método se llama cada vez que se crea una nueva instancia de la clase.

Definiendo métodos. El parámetro self

El método constructor, al igual que todos los métodos de cualquier clase, recibe como primer parámetro a la instancia sobre la que está trabajando. Por convención a ese primer parámetro se lo suele llamar self (que podríamos traducir como yo mismo), pero puede llamarse de cualquier forma.

Para referirse a los atributos de objetos hay que hacerlo a partir del objeto self.

Definición de objetos

Vamos a crear una nueva clase:

```
import math

class punto():

""" Representación de un punto en el plano, los atributos son x e y

que representan los valores de las coordenadas cartesianas."""


def __init__(self,x=0,y=0):

        self.x=x

        self.y=y


def distancia(self, otro):

        """ Devuelve la distancia entre ambos puntos. """

        dx = self.x - otro.x

        dy = self.y - otro.y

        return math.sqrt((dx*dx + dy*dy))
```

Para crear un objeto, utilizamos el nombre de la clase enviando como parámetro los valores que va a recibir el constructor.

```
>>> punto1=punto()

>>> punto2=punto(4,5)

>>> print(punto1.distancia(punto2))

6.4031242374328485
```

Podemos acceder y modificar los atributos de objeto:

```
>>> punto2.x

4

>>> punto2.x = 7

>>> punto2.x

7
```

Conceptos avanzados de programación orientada a objetos I

Atributos de clase (estáticas)

En Python, las variables definidas dentro de una clase, pero no dentro de un método, son llamadas variables estáticas o de clase. Estas variables son compartidas por todos los objetos de la clase.

Para acceder a una variable de clase podemos hacerlo escribiendo el nombre de la clase o a través de self.

```
>>> class Alumno():

...    contador=0

...    def __init__(self,nombre=""):

...       self.nombre=nombre

...       Alumno.contador+=1

...

>>> a1=Alumno("jose")

>>> a1.contador

1

>>> Alumno.contador

1
```

Usamos las variables estáticas (o de clase) para los atributos que son comunes a todos los atributos de la clase. Los atributos de los objetos se definen en el constructor.

Atributos privados y ocultos

Las variables que comienzan por un guión bajo _ son consideradas privadas. Su nombre indica a otros programadores que no son públicas: son un detalle de implementación del que no se puede depender — entre otras cosas porque podrían desaparecer en cualquier momento. **Pero nada nos impide acceder a esas variables.**

```
>>> class Alumno():
...     def __init__(self,nombre=""):
...         self.nombre=nombre
...         self._secreto="asdasd"
...
>>> a1=Alumno("jose")
>>> a1.nombre
'jose'
>>> a1._secreto
'asdasd'
```

Dos guiones bajos al comienzo del nombre __ llevan el ocultamiento un paso más allá, "enmaráñando" (name-mangling) la variable de forma que sea más difícil verla desde fuera.

```
>>> class Alumno():
...     def __init__(self,nombre=""):
...         self.nombre=nombre
...         self.__secreto="asdasd"
...
>>> a1=Alumno("jose")
>>> a1.__secreto
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Alumno' object has no attribute '__secreto'
```

Pero en realidad sigue siendo posible acceder a la variable.

```
>>> a1._Alumno__secreto

'asdasd'
```

Se suelen utilizar cuando una subclase define un atributo con el mismo nombre que la clase madre, para que no coincidan los nombres.

Métodos de clase (estáticos)

Los métodos estáticos (static methods) son aquellos que no necesitan acceso a ningún atributo de ningún objeto en concreto de la clase.

```
>>> class Calculadora():
...   def __init__(self,nombre):
...     self.nombre=nombre
...   def modelo(self):
...     return self.nombre
...   @staticmethod
...   def sumar(x,y):
...     return x+y
...
>>> a=Calculadora("basica")
>>> a.modelo()
'basica'
>>> a.sumar(3,4)
7
>>> Calculadora.sumar(3,4)
7
```

Nada nos impediría mover este método a una función fuera de la clase, ya que no hace uso de ningún atributo de ningún objeto, pero la dejamos dentro porque su lógica (hacer sumas) pertenece conceptualmente a Calculadora.

Lo podemos llamar desde el objeto o desde la clase.

Funciones getattr,setattr,delattr,hasattr

```
>>> a1=Alumno("jose")

>>> getattr(a1,"nombre")

'jose'

>>> getattr(a1,"edad","no tiene")

'no tiene'


>>> setattr(a1,"nombre","pepe")

>>> a1.nombre

'pepe'


>>> hasattr(a1,"nombre")

True


>>> delattr(a1,"nombre")
```

Conceptos avanzados de programación orientada a objetos II

Propiedades: getters, setters, deleter

Para implementar la encapsulación y no permitir el acceso directo a los atributos, podemos poner los atributos ocultos y declarar métodos específicos para acceder y modificar los atributos (mutadores). Estos métodos se denominan getters y setters.

```
class circulo():

    def __init__(self,radio):

        self.set_radio(radio)

    def set_radio(self,radio):
```

```
            if radio>=0:
                self._radio = radio
            else:
                raise ValueError("Radio positivo")
                self._radio=0
    def get_radio(self):
            print("Estoy dando el radio")
            return self._radio


>>> c1=circulo(3)
>>> c1.get_radio()
Estoy dando el radio
3
>>> c1.set_radio(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jose/github/curso_python3/curso/u51/circulo.py", line 8, in set_radio
    raise ValueError("Radio positivo")
ValueError: Radio positivo
```

En Python, las propiedades nos permiten implementar la funcionalidad exponiendo estos métodos como atributos.

```
class circulo():
    def __init__(self,radio):
            self.radio=radio


    @property
    def radio(self):
            print("Estoy dando el radio")
```

```
            return self._radio

    @radio.setter
    def radio(self,radio):
            if radio>=0:
                    self._radio = radio
            else:
                    raise ValueError("Radio positivo")
                    self._radio=0
```

```
>>> c1=circulo(3)
>>> c1.radio
Estoy dando el radio
3
>>> c1.radio=4
>>> c1.radio=-1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line 15, in radio
    raise ValueError("Radio positivo")
ValueError: Radio positivo
```

Hay un tercera property que podemos crear: el deleter

...

```
@radio.deleter
def radio(self):
        del self._radio
```

```
>>> c1=circulo(3)

>>> c1.radio

Estoy dando el radio

3

>>> del c1.radio

>>> c1.radio

Estoy dando el radio

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line 8, in radio

    return self._radio

AttributeError: 'circulo' object has no attribute '_radio'

>>> c1.radio=3
```

Representación de objetos __str__ y __repr__

La documentación de Python hace referencia a que el método __str()__ ha de devolver la representación "informal" del objeto, mientras que __repr()__ la "formal".

- La función __str()__ debe devolver la cadena de texto que se muestra por pantalla si llamamos a la función str(). Esto es lo que hace Python cuando usamos print. Suele devolver el nombre de la clase.
- De __repr()__, por el otro lado, se espera que nos devuelva una cadena de texto con una representación única del objeto. Idealmente, la cadena devuelta por __repr()__ debería ser aquella que, pasada a eval(), devuelve el mismo objeto.

Continuamos con la clase circulo:

```
...

def __str__(self):

        clase = type(self).__name__

        msg = "{0} de radio {1}"

        return msg.format(clase, self.radio)
```

```
def __repr__(self):
        clase = type(self).__name__
        msg = "{0}({1})"
        return msg.format(clase, self.radio)
```

Suponemos que estamos utilizando la clase circulo sin la instrucción print en el getter.

```
>>> c1=circulo(3)
>>> print(c1)
circulo de radio 3
>>> repr(c1)
'circulo(3)'
>>> type(eval(repr(c1)))
<class 'circulo2.circulo'>
```

Comparación de objetos __eq__

Tampoco podemos comparar dos circulos sin definir __eq()__, ya que sin este método Python comparará posiciones en memoria.

Continuamos con la clase circulo:

```
...
def __eq__(self,otro):
        return self.radio==otro.radio
```

```
>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 == c2
False
```

Si queremos utilizar <, <=, > y >= tendremos que rescribir los métodos: __lt()__, __le()__, __gt()__ y __ge()__

Operar con objetos __add__ y __sub__

Si queremos operar con los operadores + y -:

```
def __add__(self,otro):
        self.radio+=otro.radio


def __sub__(self,otro):
        if self.radio-otro.radio>=0:
                self.radio-=otro.radio
        else:
                raise ValueError("No se pueden restar")
```

```
>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 + c2
>>> c1.radio
8
```

```
>>> c1=circulo(5)
>>> c2=circulo(3)
>>> c1 - c2
>>> c1.radio
2
>>> c1 - c2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "/home/jose/github/curso_python3/curso/u52/circulo2.py", line 42, in __sub__

    raise ValueError("No se pueden restar")

ValueError: No se pueden restar
```

Más métodos especiales

Existen muchos más métodos especiales que podemos sobreescribir en nuestras clases para añadir funcionalidad a las mismas. Puedes ver la documentación oficial para aprender más sobre ellas.

Polimorfismo, herencia y delegación

Polimorfismo

El polimorfismo es la técnica que nos posibilita que al invocar un determinado método de un objeto, podrán obtenerse distintos resultados según la clase del objeto. Esto se debe a que distintos objetos pueden tener un método con un mismo nombre, pero que realice distintas operaciones.

Lo llevamos usando desde principio del curso, por ejemplo podemos recorrer con una estructura for distintas clases de objeto, debido a que el método especial __iter__ está definido en cada una de las clases. Otro ejemplo sería que con la función print podemos imprimir distintas clases de objeto, en este caso, el método especial __str__ está definido en todas las clases.

Además esto es posible ya que python es dinámico, es decir en tiempo de ejecución es cuando se determina el tipo de un objeto. Veamos un ejemplo:

```
class gato():

        def hablar(self):

                print("MIAU")


class perro():

        def hablar(self):

                print("GUAU")
```

```
def escucharMascota(animal):
    animal.hablar()


if __name__ == '__main__':
    g = gato()
    p = perro()
    escucharMascota(g)
    escucharMascota(p)
```

Herencia

La herencia es un mecanismo de programación orientada a objetos que sirve para crear clases nuevas a partir de clases preexistentes. Se toman (heredan) atributos y métodos de las clases viejas y se los modifica para modelar una nueva situación.

La clase desde la que se hereda se llama clase base y la que se construye a partir de ella es una clase derivada.

Si nuestra clase base es la clase punto estudiadas en unidades anteriores, puedo crear una nueva clase de la siguiente manera:

```
class punto3d(punto):
    def __init__(self,x=0,y=0,z=0):
        super().__init__(x,y)
        self.z=z
    @property
    def z(self):
        return self._z


    @z.setter
    def z(self,z):
        self._z=z


    def __str__(self):
```

```
        return super().__str__()+":"+str(self.z)


    def distancia(self,otro):
        dx = self.x - otro.x
        dy = self.y - otro.y
        dz = self.z - otro.z
        return (dx*dx + dy*dy + dz*dz)**0.5
```

Creemos dos objetos de cada clase y veamos los atributos y métodos que tienen definido:

```
>>> p=punto(1,2)
>>> dir(p)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_x', '_y', 'distancia', 'x', 'y']
>>> p3d=punto3d(1,2,3)
>>> dir(p3d)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_x', '_y', '_z', 'distancia', 'x', 'y', 'z']
```

La función super()

La función super() me proporciona una referencia a la clase base. Y podemos observar también que hemos reescrito el método distancia y __str__.

```
>>> p.distancia(punto(5,6))
5.656854249492381
>>> p3d.distancia(punto3d(2,3,4))
1.7320508075688772
>>> print(p)
1:2
```

```
>>> print(p3d)

1:2:3
```

Herencia múltiple

La herencia múltiple se refiere a la posibilidad de crear una clase a partir de múltiples clases superiores. Es importante nombrar adecuadamente los atributos y los métodos en cada clase para no crear conflictos.

```
class Telefono:

    "Clase teléfono"

    def __init__(self,numero):

        self.numero=numero

    def telefonear(self):

        print('llamando')

    def colgar(self):

        print('colgando')

    def __str__(self):

        return self.numero


class Camara:

    "Clase camara fotográfica"

    def __init__(self,mpx):

        self.mpx=mpx

    def fotografiar(self):

        print('fotografiando')

    def __str__(self):

        return self.mpx

class Reproductor:

    "Clase Reproductor Mp3"

    def __init__(self,capcidad):
```

```
        self.capacidad=capcidad
    def reproducirmp3(self):
        print('reproduciendo mp3')
    def reproducirvideo(self):
        print('reproduciendo video')
    def __str__(self):
        return self.capacidad


class Movil(Telefono, Camara, Reproductor):
    def __init__(self,numero,mpx,capacidad):
        Telefono.__init__(self,numero)
        Camara.__init__(self,mpx)
        Reproductor.__init__(self,capacidad)
    def __str__(self):
                                        return "Número: {0}, Cámara: {1},Capacidad: {2}".format(Telefono.__str__(self),Camara.__str__(self),Reproductor.__str__(self))
```

Veamos los atributos y métodos de un objeto Movil:

```
>>> mimovil=Movil("645234567","5Mpx","1G")
>>> dir(mimovil)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'capacidad', 'colgar', 'fotografiar', 'mpx', 'numero', 'reproducirmp3', 'reproducirvideo', 'telefonear']
>>> print(mimovil)
Número: 645234567, Cámara: 5Mpx,Capacidad: 1G
```

Funciones issubclass() y isinstance()

La función issubclass(SubClase, ClaseSup) se utiliza para comprobar si una clase (SubClase) es hija de otra superior (ClaseSup), devolviendo True o False según sea el

caso.

```
>>> issubclass(Movil,Telefono)

True
```

La función booleana isinstance(Objeto, Clase) se utiliza para comprobar si un objeto pertenece a una clase o clase superior.

```
>>> isinstance(mimovil,Movil)

True
```

Delegación

Llamamos delegación a la situación en la que una clase contiene (como atributos) una o más instancias de otra clase, a las que delegará parte de sus funcionalidades.

A partir de la clase punto, podemos crear la clase circulo de esta forma:

```
class circulo():


    def __init__(self,centro,radio):

        self.centro=centro

        self.radio=radio


    def __str__(self):

        return "Centro:{0}-Radio:{1}".format(self.centro.__str__(),self.radio)
```

Y creamos un objeto circulo:

```
>>> c1=circulo(punto(2,3),5)

>>> print(c1)

Centro:2:3-Radio:5
```