

Introducción a python3

TEMA 5. SISTEMAS DE GESTIÓN EMPRESARIAL

Características de Python

Python es un lenguaje:

- Interpretado
- Alto nivel
- Multiparadigma
- Multiplataforma
- Libre

¿Por qué elegir python?

- Porque es fácil de aprender
- Sintaxis muy limpia y sencilla
- Hay que escribir menos
- Obtienes resultados muy rápido
- Puedes programar con distintos paradigmas:
- Puedes programar distintos tipos de aplicaciones:
- Muchos usan Python (Google, IBM). Es demandado.
- Gran cantidad de módulos, muchísimas funcionalidades.
- Una gran comunidad que apoya el proyecto.
- Viene preinstalado en la mayoría de sistemas

Origen de Python

1991: [Guido van Rossum](#)



[*Benevolente Dictador Vitalicio*](#) (en inglés: *Benevolent Dictator for Life*, BDFL)

Entonces, ¿qué van a hacer todos ustedes? ¿Crear una democracia? ¿Anarquía? ¿Una dictadura? ¿Una federación?

Guido van Rossum⁸










¿Por qué elegir python?



About us ▾ Knowledge News Coding Standards TIOBE Index Contact 🔍

Products ▾ Quality Models ▾ Markets ▾ [Schedule a demo](#)

when starting to build a new software system. The definition of the TIOBE index can be found [here](#).

Jan 2024	Jan 2023	Change	Programming Language		Ratings	Change
1	1			Python	13.97%	-2.39%
2	2			C	11.44%	-4.81%
3	3			C++	9.96%	-2.95%
4	4			Java	7.87%	-4.34%
5	5			C#	7.16%	+1.43%
6	7	▲		JavaScript	2.77%	-0.11%
7	10	▲		PHP	1.79%	+0.40%
8	6	▼		Visual Basic	1.60%	-3.04%
9	8	▼		SQL	1.46%	-1.04%

Índice TIOBE

<http://www.tiobe.com/tiobe-index/>

El Zen de Python

El código que siga los principios de Python se dice que es "pythónico".

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Lo práctico gana a lo puro.
- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una —y preferiblemente solo una— manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.²⁴
- Ahora es mejor que nunca.
- Aunque *nunca* es a menudo mejor que *ya mismo*.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (*namespaces*) son una gran idea ¡Hagamos más de esas cosas!

Instalación de python3

- Instalación en linux debian/Ubuntu
- Instalación en Windows
 - Descargar instalador (paquete MSI)
- Instalación en Mac

<https://www.python.org/>

<https://www.jetbrains.com/es-es/pycharm/>

Estructura de un programa

Estructura de un 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.

```
# El punto y coma “;” se puede usar para separar varias sentencias  
# en una misma línea, pero no se aconseja su uso.
```

```
edad = 15; print(edad)
```

Estructura de un programa

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 indentación.

- La indentación se puede hacer con espacios y tabulaciones pero ambos tipos no se pueden mezclar. Se recomienda usar 4 espacios.
- Cuando el bloque a sangrar sólo ocupa una línea ésta puede escribirse después de los dos puntos.

```
if num >=0:
    while num<10:
        print (num)
        num = num +1
```

```
if azul: print('Cielo')
```

Estructura de un programa

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.

```
if condicion1 and condicion2 and condicion3 and \  
    condicion4 and condicion5:
```

```
dias = ['lunes', 'martes', 'miércoles', 'jueves',  
        'viernes', 'sábado', 'domingo']
```

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	

Comentarios

Se utiliza el carácter # para indicar los comentarios al final de la línea.


Se utiliza el triple entrecomillado para comentarios de más de una línea.

```
'''  
Comentario más largo en una línea en Python  
'''  
print("Hola mundo") # También es posible añadir un comentario al final de una línea de código
```

Escribir y ejecutar programas python3

Uso del interprete

```
C:\Users\Victoria>python
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bi
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



¿IDE o
editor de
texto?

A partir de un fichero con el código fuente en ubuntu

```
$ python3 programa.py
```

Mi primer programa en python3

Python3

```
# Programa que pida la edad
# y diga si es mayor de edad.
edad=int(input("Dime tu  edad:"))
if edad>=18:
    print("Eres mayor de edad")
print("Programa Terminado")
```

Módulos

Instrucciones almacenadas en un fichero y ejecutadas por el interprete de Python.

Crear un fichero de texto 'miprimerprograma.py'

```
x = 4
```

```
y = 5
```

```
x ** (y + 1)
```

En Linux:

```
chmod +x fichero.
```

```
#!/usr/bin/python3: Primera línea del módulo: especifica que debe utilizarse Python para ejecutar el código contenido en el fichero (Linux).
```

Ejecucción:

```
$ python3 ejemplo.py
```

```
$ ./ejemplo.py
```


Funciones predefinidas

Tenemos una serie de funciones predefinidas en python3:

<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

Todas estas funciones y algunos elementos comunes del lenguaje están definidas en el módulo [`builtins`](#).

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.

<https://docs.python.org/es/3.12/index.html>

Tipos de datos básicos

Literales, variables y expresiones

Literales: valores constantes

- **Literales enteros:** 3, 12, -23
- **Literales reales:** 12.3, 45.6
- **Literales cadenas:**
 'hola que tal!'
 "Muy bien"
 '''Podemos \n
 ir al cine'''

Variable: Referencia un valor

Variables:

```
>>> var = 5  
>>> var 5
```

Expresiones: combinación entre literales y variables

Expresiones

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

Operadores

- Operadores aritméticos: `+`, `-`, `*`, `/`, `//`, `%`, `**`.
- Operadores de cadenas: `+`, `*`
- Operadores de asignación: `=`
- Operadores de comparación: `==`, `!=`, `>=`, `>`, `<=`, `<`
- Operadores lógicos: `and`, `or`, `not`
- Operadores de pertenencia: `in`, `not in`

Precedencia de operadores

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

Tipos de datos

númericos

Tipo entero
(int)

Tipo real
(float)

Tipo numérico
(complex)

booleanos
(bool)

secuencia

Tipo lista
(list)

Tipo tuplas
(tuple)

Tipo rango
(range)

cadenas de
caracteres

Tipo cadena
(str)

Tipos de datos

binarios

Tipo byte

Tipo
bytearray

conjuntos

(set)

(frozenset)

iterador y
generador (iter)

mapas o
diccionario (dict)

Funciones tipos de datos

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
```

Tipos 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.

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

Tipos de datos numéricos

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).

```
>>> a=int(7.2)
>>> a
7
>>> a=int("345")
>>> a
345
>>> b=float(1)
>>> b
1.0
>>> =float("1.234")
>>> b
1.234
>>> a=int("123.3")
```

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.

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

```
>>> import math
>>> math.sqrt(9)
3.0
```

Tipo de datos booleanos

True, False

¿Qué valores se consideran falsos?

- False
- Cualquier número 0. (0, 0.0)
- Cualquier secuencia vacía (`[]`, `()`, `"`)
- Cualquier diccionario vacío (`{}`)

Operadores Lógicos

```
x or y  
x and y  
not x
```

Operadores de Comparación

```
== != >= > <= <
```

Variables

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'>
```

Trabajando con variables

Una variables es un identificador que referencia a un valor. 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.

```
>>> a = 5
>>> a
5
>>> del a
>>> a
Traceback (most recent call
last):
  File "<stdin>", line 1, in
<module>
```

```
>>> a = 5
>>> a
5
>>> a = 8
>>> a
8
>>> a = a + 1
>>> a+=1
>>> 10
```

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.

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


Objetos mutables e inmutables

Objetos inmutables

Ciertos objetos son inmutables, es decir, no pueden modificar su valor.

```
>>> a = "hola"
```

```
>>> a[0]="m"
```

```
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

Se puede modificar un elemento de una lista.

```
>>> a = [1,2]
```

```
>>> b = a
```

```
>>> a[0] = 5
```

```
>>> b
```

```
[5, 2]
```

De los tipos de datos principales, hay que recordar que son mutables son las listas y los diccionarios.

Operadores de asignación

Permiten asignar una valor a una variable, o mejor dicho: 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"
```

Operadores relacionales y lógicos

Operadores relacionales

```
>>> 2 == 3
```

```
False
```

```
>>> 2 != 3
```

```
True
```

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

```
False
```

```
>>> 5 > 5
```

```
False
```

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

```
True
```

```
>>> not 2 > 3
```

```
True
```

```
>>> x = 1.56
```

```
>>> x >= 0 and x <= 1
```

```
False
```

```
>>> x = 1.56
```

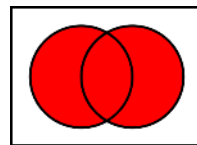
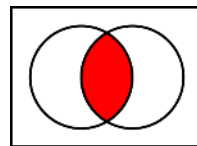
```
>>> x >= 0 or x <= 1
```

```
True
```

```
>>> x = 1.56
```

```
>>> 0 <= x <= 1
```

```
False
```



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.

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

Entrada y salida estándar

Función print

- No permite escribir en la salida estándar.
- Podemos indicar varios datos a imprimir, que por defecto serán separado 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.

```
>>> print(1,2,3)
```

```
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
```

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

```
2 2.500000 2.5
```

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

```
El producto cesta cantidad=23 precio=13.46
```

Fomateando cadenas de caracteres

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.

```
>>> print(format(31,"b"),format(31,"o"),format(31,"x"))
```

```
11111 37 1f
```

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

```
2.35
```

Ejemplos print()

```
>>> prin'Hola {}'.format("Juan")  
'Hola Juan'
```

```
>>> 'Hola {}'.format(3.14)  
'Hola 3.14'
```

```
>>> 'Hola {} y {}'.format('Juan',  
'Mar')  
'Hola Juan y Mar'
```

```
>>> 'Hola {1} y {0}'.format('J.', 'M.')  
'Hola M. y J.'
```

```
>>> '7 / 3 = {}'.format(7.0/3)  
'7 / 3 = 2.333333333333'
```

```
>>> '7 / 3 = {:.2f}'.format(7.0/3)  
'7 / 3 = 2.33'
```

<https://docs.python.org/es/3/tutorial/inputoutput.html>

Cadenas "f"

En Python 3.6 se añadió (PEP 498) una nueva notación para cadenas llamada cadenas "f", que simplifica la inserción de variables y expresiones en las cadenas. Una cadena "f" contiene variables y expresiones entre llaves ({}), que se sustituyen directamente por su valor. Las cadenas "f" se reconocen porque comienzan por una letra f antes de las comillas de apertura.

```
nombre = "Pepe"
```

```
edad = 25
```

```
print(f"Me llamo {nombre} y tengo {edad} años.")
```

Me llamo Pepe y tengo 25 años.

```
semanas = 4
```

```
print(f"En {semanas} semanas hay {7 * semanas} días.")
```

En 4 semanas hay 28 días.

Estructuras de control

Estructura de control: Alternativas

Alternativa simple

```
edad = int(input("Dime tu
edad:"))
if edad >= 18:

    print("Eres mayor de ed
print("Programa terminado")
```

Alternativa doble

```
edad = int(input("Dime tu
edad:"))
if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
print("Programa terminado")
```

Estructura de control: Alternativas

```
nota = int(input("Dime tu nota:"))
if nota >=1 and nota <= 4:
    print("Suspenso")
elif nota == 5:
    print("Suficiente")
elif nota == 6 or nota == 7:
    print("Bien")
elif nota == 8:
    print("Notable")
elif nota ==9 or nota == 10:
    print("Sobresaliente")
else:
    print("Nota incorrecta")
print("Programa terminado")
```

Estructura de control: Alternativas

A partir de la versión 3.10 de Python existe la sentencia match como alternativa al switch.

```
status= int (input("Introduce el error"))
match status:
    case 400:
        print("Bad request")
    case 404:
        print("Not found")
    case 418:
        print("I'm a teapot")
    case 401 | 403 | 404:
        print("Not allowed")
    case _:
        print("Something's wrong with the internet")
```

Ejemplos

https://colab.research.google.com/drive/166sF7rYKsXPOy5j4Y2Iglhme_r-59aKr?usp=sharing

Estructura de control repetitivas: while

while

```
secreto = "asdasd"
clave = input("Dime la clave:")
while clave != secreto:
    print("Clave incorrecta!!!")
    clave = input("Dime la clave:")
print("Bienvenido!!!")
print("Programa terminado")
```

Instrucciones: continue

```
cont = 0
while cont < 10:
    cont = cont + 1
    if cont % 2 != 0:
        continue
    print(cont)
```

Instrucciones: break

```
secreto = "asdasd"
clave = input("Dime la clave:")
while clave != secreto:
    print("Clave incorrecta!!!")
    otra = input("¿Quieres introducir otra clave (S/N)?:")
    if otra.upper() == "N":
        break;
    clave = input("Dime la clave:")
if clave == secreto:
    print("Bienvenido!!!")
print("Programa terminado")
```

Estructura de control repetitivas: while

```
secreto = "asdasd"
while True:
    clave = input("Dime la clave:")
    if clave != secreto:
        print("Clave incorrecta!!!")
    if clave == secreto:
        break;
print("Bienvenido!!!")
print("Programa terminado")
```

¿Y la estructura
“Repetir” de
pseudocódigo?

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.

Estructura 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")
```

Estructuras de control

for y range

```
for i in range(10):  
    print i
```

No utilizamos condiciones de inicio, parada e incremento, sino que especificamos claramente qué elementos se utilizan

Más expresivo y seguro

range(n)

```
>>> x= list(range(7))  
[0, 1, 2, 3, 4, 5, 6]
```

Genera una lista de n valores [0, 1, 2 ... n-1]

range(start, n)

```
>>> x= list( range(3, 7) )  
[3, 4, 5, 6]
```

Genera la lista [start, start+1, ... n-1]

range(start, n, step)

```
>>> x= list( range(1, 11, 2))  
[1, 3, 5, 7, 9]
```

Genera la lista [start, start+step, ... n-1]

range(start, n)

```
>>> x= list( range(3, 7))  
[3, 4, 5, 6]
```

Genera la lista [start, start+1, ... n-1]

Estructuras de control

for y range

```
>>> for i in range(3):  
...     print "hola"  
...  
hola  
hola  
hola
```

Estructuras de control

for y range

```
>>> for letra in "Sara":  
...     print letra  
...  
S  
a  
r  
a
```

Estructuras de control

for y range

```
>>> numeros = [2, 4, 5]
```

index 0 → 2

index 1 → 4

index 2 → 5

¿Y si dentro del bucle necesitamos conocer el índice del elemento?

```
>>> numeros = [2, 4, 5]
```

```
>>> for i in range(len(numeros)):
```

```
...     print i, '->', numeros[i]
```

Estructuras de control

for y enumerate(sec)

```
>>> for i, num in enumerate([1, 2, 3]):  
...     print (i, "->", num)
```

Permite iterar simultáneamente sobre el índice y sobre el elemento de una secuencia.

La función incorporada `enumerate()` toma como argumento un objeto iterable `it` y retorna otro cuyos elementos son tuplas de dos objetos, el primero de los cuales indica la posición de un elemento perteneciente a `it` y el segundo, el elemento mismo.

Estructuras de control

for y enumerate(sec)

```
>>> lenguajes = ["Java", "C", "C++", "Rust", "Elixir"]
>>> list(enumerate(lenguajes))
[(0, 'Java'), (1, 'C'), (2, 'C++'), (3, 'Rust'), (4, 'Elixir')]
>>> for i, lenguaje in enumerate(lenguajes):
...     print(i, lenguaje)
.
0 Java
1 C
2 C++
3 Rust
4 Elixir
```

Usos específico de variables: contadores

```
cont = 0; ← El contador se inicializa a un valor inicial
for var in range(1, 6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        cont = cont + 1 ← El contador se incrementa
                           Un contador también se puede decrementar
print("Has introducido ", cont, " números pares.")
```

Un contador es una variable entera que la utilizamos para contar cuando ocurre un suceso.

Usos específico de variables: acumuladores

```
suma = 0;
for var in range(1, 6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        suma = suma + num
print("La suma de los números pares es ", suma)
```

El acumulador se **inicializa** a un valor inicial

El acumulador se **acumula**
Podemos acumular también **productos**

Un acumulador es una variable numérica que permite ir acumulando operaciones. Me permite ir haciendo operaciones parciales.

Usos específico de variables: indicadores

```
indicador = False;
```

Se **inicializa** a un valor lógico:

No ha sucedido!!!!

```
for var in range(1,6):
```

```
    num = int(input("Dime un número:"))
```

```
    if num % 2 == 0:
```

```
        indicador = True
```

Cuando ocurre el suceso **cambiamos** su valor

```
if indicador:
```

```
    print("Has introducido algún número par")
```

```
else:
```

```
    print("No has introducido algún número par")
```

Al final debemos
comprobar **si el suceso
ha ocurrido**

Un indicador es una variable lógica, que usamos para recordar o indicar algún suceso.

Ejemplos bucles

<https://colab.research.google.com/drive/1MbLOZKcMRU2KmQ96fjGWLnKgHOgma2ZC?usp=sharing>

Tipos de datos secuencias

Cadenas de caracteres

```
>>> 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])
```

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

Cadenas de caracteres

- Las cadenas se pueden recorrer.
- Operadores de pertenencia: in y not in.
- Concatenación: +
- Repetición: *
- Indexación
- Slice
- funciones definidas: len, max, min, sorted.

+: Concatenación

```
>>> "hola " + "que tal"  
'hola que tal'
```

***: Repetición**

```
>>> "abc" * 3  
'abcbcabcb'
```

Indexación

```
>>> cadena = "josé"  
>>> cadena[0]  
'j'  
>>> cadena[3]  
'é'
```

Longitud

```
>>> cadena = "josé"  
>>> len(cadena)  
4
```

Operadores

Comparación

```
"a">"A"  
True  
  
"informatica">"informacion"  
"  
True  
  
"abcde">"abcdef"  
False
```


Cadenas de caracteres

Comillas simples o dobles:

```
>>> nombre = 'Sara'
```

```
>>> nombre
```

```
'Sara'
```

```
>>> frase = 'Sara dijo "hola"'
```

```
>>> frase
```

```
'Sara dijo "hola"'
```

Secuencia de escape:

```
"Sara dijo \"hola\""
```

Triple entrecomillado:

```
"""Sara dijo
```

```
"hola" """
```

Cadenas de caracteres

Slices

a[index]: Devuelve el elemento index

```
>>> a = "Abstulit qui dedit"
```

```
a[0] → "A"
```

```
a[6] → "i"
```

```
a[-1] → "t"
```

a[start:end]: Elementos desde start hasta end-1 inclusive.

```
a = "Alea jacta est"
```

```
a[0:10] → "Alea jacta"
```

```
a[5:10] → "jacta"
```

```
a[5:6] → "j"
```

```
a[5:5] → ""
```

Cadenas de caracteres

Slices

a[start:] Elementos desde *start* hasta el final.

a = "Fabricando fit faber"

a[12:] → "fit faber"

a[-5:] → "faber"

a[:N] N elementos, desde el inicio hasta N-1 inclusive.

a = "In vino veritas"

a[:2] → "In"

a[:8] → "In vino"

a[:-5] → "In vino ve"

Cadenas de caracteres

Slices

`a[start:end:step]` De *start* a *end* de *step* en *step* elementos.

```
a = "Is fecit, cui prodest"
```

```
a[::2] = "I ei,cipoet"
```

```
a[:12:3] = "Ifi "
```

```
a[::-1] = "tsedorp iuc ,ticef sl"
```

Cadenas de caracteres

Inmutables

```
>>> nombre = "Sara"
```

```
>>> nombre[0] = "M"
```

```
TypeError: 'str' object does not support item assignment
```

```
>>> nombre = "Sara"
```

```
>>> nombre = "Mara"
```

```
>>> nombre
```

```
'Mara'
```

```
>>> nombre = "Sara"
```

```
>>> nombre = "M" + nombre[1:] Concatenación
```

```
>>> nombre
```

```
'Mara'
```

Cadenas de caracteres

Métodos de la clase str

`cadena.capitalize`

`cadena.casefold`

`cadena.center`

`cadena.count`

`cadena.encode`

`cadena.endswith`

`cadena.expandtabs`

`cadena.find`

`cadena.format`

`cadena.format_map`

`cadena.index`

`cadena.isalnum`

`cadena.isalpha`

`cadena.isdecimal`

`cadena.isdigit`

`cadena.isidentifier`

`cadena.islower`

`cadena.isnumeric`

`cadena.isprintable`

`cadena.isspace`

`cadena.istitle`

`cadena.isupper`

`cadena.join`

`cadena.ljust`

`cadena.lower`

`cadena.lstrip`

`cadena.maketrans`

`cadena.partition`

`cadena.replace`

`cadena.rfind`

`cadena.rindex`

`cadena.rjust`

`cadena.rpartition`

`cadena.rsplit`

`cadena.rstrip`

`cadena.split`

`cadena.splitlines`

`cadena.startswith`

`cadena.strip`

`cadena.swapcase`

`cadena.title`

`cadena.translate`

`cadena.upper`

`cadena.zfill`

Cadenas de caracteres

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,"="))
```

```
=====h  
ola mundo
```

```
>>> num = 123
```

```
>>> print(str(num).zfill(12))
```

```
0000000000123
```

Cadenas de caracteres

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
```

```
>>> "ere" in "sapere aude"
```

```
True
```

```
>>> "sapere aude".find("ere")
```

```
3
```

La subcadena existe y empieza en el elemento 3 (es decir, el cuarto)

```
>>> "flux" in "et veritas"
```

```
False
```

```
>>> "et veritas".find("flux")
```

```
-1
```

La subcadena no existe

Cadenas de caracteres

Métodos de validación

```
cad = "bienvenido a mi aplicación"
```

```
>>> 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()`,...

Cadena de caracteres

Otras funciones

```
>>> len("victoria aut mors")
```

```
17
```

Tamaño

```
>>> comic = "V de Vendetta"
```

```
>>> len(comic)
```

```
13
```

```
>>> "hoygan".upper()
```

```
"HOYGAN"
```

Mayúsculas

```
>>> vocales = "AEIOU"
```

```
>>> vocales.lower()
```

```
"aeiou"
```

Minúsculas

Cadenas de caracteres

Eliminando espacios en blanco

```
>>> planeta = "    Saturno"
>>> planeta.lstrip()
"Saturno"
>>> lugar = "En el Sol    "
>>> print lugar.rstrip(), "hace
calor"
En el Sol hace calor
>>> apellido = "    Lee    "
>>> print "Bruce", apellido.strip(),
"Jr."
Bruce Lee Jr.
```

```
>>> cadena = "
www.eugeniabahit.com  "
>>> print(cadena.strip())
www.eugeniabahit.com
>>>
cadena="00000000012300000000
0"
>>> print(cadena.strip("0"))
123
```

Métodos principales de cadenas

Métodos de sustitución

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

Estimado Sr. Juan Pérez:

Métodos de unión y división

```
>>> hora = "12:23:12"
>>> print(hora.split(":"))
['12', '23', '12']

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

Proceso de cadenas

split

`split()`

Divide una cadena de texto en una serie de palabras, utilizando espacios en blanco y saltos de línea como delimitador.

```
>>> linea = "esto es un fichero\n"
>>> linea.split()
['esto', 'es', 'un', 'fichero']

>>> numeros = "1 2 3"
>>> numeros.split()
['1', '2', '3'] # lista de str!
>>> x = int(numeros.split()[-1])
>>> print x ** 2
9
```

Proceso de cadenas

join

`join()`

Une las cadenas de texto de una lista en una única cadena, separadas por el delimitador especificado.

```
>>> numeros = "1 2 3"
>>> numeros.split()
['1', '2', '3'] # lista de str!
>>> numeros2 = "+".join(numeros.split())
>>> print numeros2
'1+2+3'
```

Ejercicios para practicar

1. Dada la cadena "Hola mundo", determina su longitud.
2. Muestra los últimos cuatro caracteres de la cadena "Programación en Python".
3. Imprime por pantalla los caracteres en las posiciones impares de la cadena "abcdefghijkl".
4. En mayúscula, muestra los caracteres en posiciones múltiplo de cinco de la cadena "".
5. De tres en tres, imprime del carácter en la posición 2 al de la posición 20 de la cadena "Ejercicios de Python".
6. Verifica si la cadena "información" contiene la subcadena "ma".
7. Determina si la cadena "Python es divertido" comienza con la palabra "Python".
8. ¿Está presente la letra "z" en la cadena "Caza zorros en la zona"?
9. ¿La cadena "Examen final" contiene la palabra "final" en mayúscula o minúscula?
10. Divide la cadena "Análisis de datos" en palabras y únelas con guiones bajos.
11. Cuenta las consonantes en la cadena "Inteligencia artificial".
12. Reemplaza las letras "a" por "o" en la cadena "La manzana está en la mesa".

Tipo de datos: Listas

- Serie de elementos separados por comas, encerrados entre corchetes.
- Los elementos pueden ser de distinto tipo
- Dinámicas: número variable de elementos
- Mutables

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


Tipo de datos secuencia: listas

Operadores

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

Recorrido

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

```
>>> lista2 = ["a","b","c","d","e"]
>>> for num,letra in zip(lista,lista2):
...     print(num,letra)
1 a
2 b
...
```

Operadores de pertenencia

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

Concatenación (+)

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

Repeteción (*)

```
>>> lista * 2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Indexación

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

```
>>> lista1[12]
...
IndexError: list index out of range
```

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

Tipo de datos secuencia: listas

Operadores

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

Slice

```
>>> lista[2:4]
[3, 4]
>>> lista[1:4:2]
[2, 4]
>>> lista[:5]
[1, 2, 3, 4, 5]
>>> lista[5:]
[6, 1, 2, 3, 4, 5, 6]
>>> lista[::-1]
[6, 5, 4, 3, 2, 1, 6, 5, 4, 3, 2, 1]
```

Listas multidimensionales

```
>>> 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()
```

Funciones

```
>>> 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]
```

Las listas son mutables

Los elementos de las listas se pueden modificar:

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

Los métodos de las listas modifican el contenido de la lista:

```
>>> lista1.append(3)
>>> lista1
[1, 2, 3]
```

¿Cómo se copian las listas?

Para copiar una lista en otra no podemos utilizar el operador de asignación:

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

El operador de asignación no crea una nueva lista, sino que nombra con dos nombres distintos a la misma lista, por lo tanto la forma más fácil de copiar una lista en otra es:

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

Métodos principales de listas

Métodos de inserción

```
>>> 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

```
>>> lista.pop()
6
Borrado por valor (remove)

>>> lista
[1, 100, 2, 3, 4, 5]
Borrado por posición (pop y del)

>>> lista.pop(1)
100

>>> lista
[1, 2, 3, 4, 5]

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

>>> del lista[0]
[2, 4, 5]
```

Métodos principales de listas

Métodos de ordenación

```
>>> lista.reverse()
```

```
>>> lista
```

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

```
>>> lista.sort()
```

```
>>> lista
```

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

```
sorted(lista1,reverse=True)
```

Métodos de búsqueda

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

```
1
```

```
>>> lista.append(5)
```

```
>>> lista
```

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

```
>>> lista.index(5)
```

```
0
```

```
>>> lista.index(5,1)
```

```
4
```

Ordenación de listas

sort

```
secuencia.sort()
```

Ordena secuencia, modificándola:

```
>>> a = [5, 7, 3, -1]
```

```
>>> a.sort()
```

```
>>> a
```

```
[-1, 3, 5, 7]
```

Ordenación de listas

sorted

```
sorted(secuencia)
```

Devuelve una copia ordenada de secuencia:

```
>>> sorted([5, 7, 3, -1])  
[-1, 3, 5, 7]
```

Ordenación de listas

sorted

```
>>> robot = "Skynet"
```

```
>>> sorted(robot)
```

```
['S', 'e', 'k', 'n', 't',  
'y']
```

```
>>> a = [1, 7, 4, 9]
```

```
>>> sorted(a, reverse=True)
```

```
[9, 7, 4, 1]
```

Argumento:

```
reverse = True
```

```
>>> lista=[1,5,7,0]
>>> lista2=sorted(lista)
>>> lista2
[0, 1, 5, 7]
>>> lista3=sorted(lista,reverse=True)
>>> lista3
[7, 5, 1, 0]
>>> lista.sort()
>>> lista
[0, 1, 5, 7]
```


Ejemplos Listas

Concatenación

```
>>> [1, 2, 3] + [9, 8]
```

```
[1, 2, 3, 9, 8]
```

```
>>> x = [1, 2, 3]
```

```
>>> x.append([9, 8])
```

```
>>> x
```

```
[1, 2, 3, [9, 8]]
```

El cuarto elemento es ahora una lista

Ejercicios para practicar

Inicializa una lista llamada `mi_lista` con al menos cinco elementos de diferentes tipos (cadena, entero, decimal).

1. ¿Cuál es el tamaño de la lista `mi_lista`?
2. Recorre la lista `mi_lista` e imprime cada elemento.
3. Calcula el tamaño de la lista multiplicado por el segundo elemento de la lista `mi_lista`.
4. Calcula el producto del segundo elemento de la lista `mi_lista` por el tercer elemento.
5. ¿Está el número 5 en la lista `mi_lista`? ¿Y el número 5.0?
6. Elimina el primer elemento de la lista `mi_lista`.
7. Elimina ahora los dos últimos elementos de la lista `mi_lista` simultáneamente.
8. ¿Está la lista `mi_lista` vacía?

Ejercicio Ordenación de listas

Crea una lista con 10 números aleatorios y realiza las siguientes operaciones:

1. El mayor número de la lista
2. El menor número de la lista
3. Los tres mayores números de la lista
4. El mayor de los 3 primeros números de la lista
5. El menor de los 4 últimos números de la lista
6. La suma de los 5 mayores números de la lista
7. La suma de los 3 menores números de la lista

Ejercicios con listas

- Crea un programa que permita al usuario ingresar las edades de 5 personas. Luego, muestra todas las edades, la edad promedio, la edad más alta y la edad más baja.
- Solicita al usuario ingresar una palabra y crea un programa que cuente la cantidad de letras y la cantidad de números presentes en la palabra.
- A partir de dos listas de cadenas, 'nombres1' y 'nombres2', crea una nueva lista que contenga aquellos nombres que tienen una longitud mayor a 5 caracteres en ambas listas. Imprime la lista resultante por pantalla.
- Dadas dos listas de números decimales, 'precios1' y 'precios2', crea una nueva lista que contenga los precios redondeados al entero más cercano. Imprime la lista resultante.
- Lee dos listas de enteros, 'numeros1' y 'numeros2', y crea un programa que determine si tienen algún número en común. Muestra un mensaje indicando si hay o no números comunes.
- Lee una cadena de texto del usuario y crea un programa que cuente la cantidad de palabras que comienzan con la letra 'a' y tienen una longitud de al menos 3 caracteres.

<https://docs.python.org/es/3/library/re.html>

Tuplas

(4, 5, 8)

Listas *inmutables*.

Paréntesis, en lugar de corchetes.

No tienen métodos (append, remove, etc).

(1,)

Definimos una tupla de un elemento

Necesaria la coma

Devolvemos múltiples valores en una función separados por comas: una tupla.

Casi siempre es preferible utilizar una lista.

```
>>> x = (1, 8)
>>> x[0]
1

>>> len(x)
2

>>> 7 in x
False
```

Tipo de datos secuencia: tuplas

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

Operadores

- **Recorrido**
- **Operadores de pertenencia**
- **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 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
```

Sets

Concepto matemático de conjunto.

Colección no ordenada y sin elementos repetidos.

Soportan operaciones matemáticas como la unión, la intersección y la diferencia.

En un conjunto los elementos no tienen posición: un elemento está o no está. Los índices y *slices* no funcionan.

```
>>> a = [1, 1, 1, 2, 2, 3, 3, 3]
```

```
>>> a = list(set(a))
```

```
>>> a
```

```
[1, 2, 3]
```

El uso más habitual es eliminar elementos duplicados

lista → set → lista

Sets

Creación

```
conjunto = set()
```

Conjunto vacío

```
conjunto = set(lista)
```

A partir de una lista

```
>>> conjunto={1,2,3,4,6,8,1}  
>>> conjunto  
{1, 2, 3, 4, 6, 8}
```

Adición

```
>>> conjunto = set()  
>>> conjunto.add(1)  
>>> conjunto.add("tres")  
>>> conjunto  
set([1, 'tres'])
```


Sets

Eliminación

```
>>> v = [1, 3, 4]
>>> v.remove(3)
>>> v
[1, 4]
>>> conjunto = set([1, 2, 3])
>>> conjunto.remove(2)
>>> conjunto
set([1, 3])
```

```
>>> conjunto
{1, 2, 3, 4, 5}
>>> conjunto.pop()
1
>>> conjunto
{2, 3, 4, 5}
>>> conjunto.pop(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: set.pop() takes no arguments (1 given)
>>> conjunto.remove(3)
>>> conjunto
{2, 4, 5}
```

Sets

Tamaño

```
>>> conjunto = set([1, 2, 3])  
>>> len(conjunto)  
3
```

Sets

Búsqueda

```
>>> conjunto = set([1, 2, 3])  
>>> 7 in conjunto  
  
False
```

Sets

Iteración

```
>>> conjunto = set([1, 2, 3])  
>>> for i in conjunto:  
...     print i  
...  
1  
2  
3
```

Sets Unión

```
>>> a = set([1, 4, 7])
>>> b = set([2, 4, 6])
>>> a.union(b)
set([1, 2, 4, 6, 7])
```

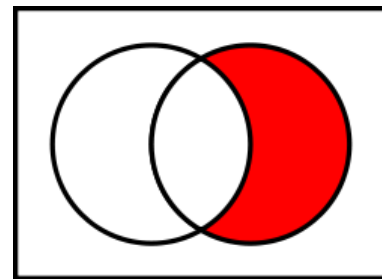
Sets Intersección

```
>>> a = set([1, 4, 7])
>>> b = set([2, 4, 6])
>>> a.intersection(b)
set([4])
```

Sets Diferencia

```
>>> a = set([1, 4, 7])
>>> b = set([2, 4, 6])
>>> a.difference(b)
set([1, 7])
```

Devuelve un nuevo conjunto con los elementos de a no presentes en b.



Diccionarios

Un conjunto de pares clave-valor, que define una relación uno a uno entre claves y valores.

- Por ejemplo, relación DNI → Persona.

A diferencia de las secuencias, donde los índices son números enteros, los diccionarios están indexados por claves.

Las claves han de ser inmutables (lo que excluye a listas y sets); los valores pueden ser de cualquier tipo.

```
diccionario = {}  
diccionario = dict()  
diccionario[clave]
```

Tipo de datos mapas: diccionarios

```
>>> diccionario = {'one': 1, 'two': 2, 'three': 3}
>>> dict1 = {}
>>> dict1["one"]=1
>>> dict1["two"]=2
>>> dict1["three"]=3
```

Operadores

Longitud

```
>>> len(a)
3
```

Indexación

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

Eliminación

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

Operadores de pertenencia

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

Los diccionarios son mutables

Los elementos de los diccionarios se pueden modificar:

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

¿Cómo se copian los diccionarios?

Para copiar un diccionario en otra no podemos utilizar el operador de asignación:

```
>>> a = {'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

Métodos de eliminación

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict1.clear()
>>> dict1
{}

```

Métodos de creación

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict2 = dict1.copy()

>>> dict1 = {'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}

```


Métodos principales de diccionarios

Métodos de retorno

```
>>> dict1 = {'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'
```

Recorridos de diccionarios

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

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

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

Diccionarios

Asignación

```
>>> vacaciones = {}  
>>> vacaciones["Sara"] = 3  
>>> vacaciones["Miguel"] = 5  
>>> vacaciones["Arcadio"] = 4  
>>> vacaciones  
{ 'Sara': 3, 'Miguel': 5, 'Arcadio': 4 }
```

Diccionarios

Extracción

```
>>> vacaciones["Sara"]  
3  
>>> vacaciones["Sara"] -= 1  
>>> vacaciones["Sara"]  
2
```

```
>>> vacaciones["Pedro"]  
KeyError: 'Pedro'
```

Diccionarios

Eliminación

```
>>> vacaciones  
{ 'Sara': 2, 'Miguel': 5, 'Arcadio': 4}  
>>> del vacaciones["Arcadio"]  
>>> vacaciones  
{ 'Sara': 2, 'Miguel': 5}
```

Diccionarios

Tamaño

```
>>> vacaciones  
{ 'Sara': 2, 'Miguel': 5}  
>>> len(vacaciones)  
2
```

Diccionarios

Búsqueda

```
>>> "Sara" in vacaciones
```

```
True
```

```
>>> 2 in vacaciones
```

```
False
```

La búsqueda se hace sobre las claves, no sobre los valores

Diccionarios

keys() y values()

```
>>> vacaciones.keys()  
['Sara', 'Miguel']  
>>> vacaciones.values()  
[2, 5]
```

Devuelven una lista con las claves y valores,
respectivamente.

```
>>> 2 in vacaciones.values()  
True
```

Diccionarios

Iteración

```
>>> for k, v in vacaciones.items():
```

```
...     print k, v
```

```
...
```

```
Sara 2
```

```
Miguel 5
```

Ejercicios para practicar

$x = [7, 3, 2, 3, 1]$

$y = [7, 2, 3, 2, 9]$

1. ¿Cuántos elementos hay en x si se eliminan los repetidos?
2. Una lista que contenga la concatenación de ambas listas.
3. Una lista que contenga la unión de ambas listas, sin duplicados.
4. Un conjunto que tenga la intersección de ambas listas.
5. Un diccionario en el que para cada entero de la lista x se almacena su cuadrado.
6. Un diccionario en el que se almacena el número de veces que cada entero aparece en la lista y .

EJERCICIO CONJUNTOS

Imagina que estás organizando un evento y deseas gestionar la lista de asistentes. Cada asistente tiene un código único y puede pertenecer a una o más categorías (por ejemplo, "VIP", "Estudiante", "Patrocinador", etc.).

Crea un programa en Python que realice las siguientes acciones:

1. Define dos conjuntos: uno para los asistentes al evento y otro para las categorías disponibles.
2. Permite al usuario ingresar códigos de asistentes y asignarlos a una o más categorías. Asegúrate de manejar casos en los que un asistente ya tenga asignada una categoría y se le quiera asignar otra.
3. Muestra la lista de asistentes y sus respectivas categorías.
4. Proporciona la funcionalidad para que el usuario pueda consultar la lista de asistentes en una categoría específica.
5. Implementa la opción para eliminar un asistente de la lista y, por ende, de todas sus categorías.

Ejercicios para practicar

Escribe un programa en Python que implemente la gestión de clientes y asistentes a eventos. Cada cliente tiene un código único, y puede pertenecer a una categoría determinada. Además, se puede buscar a los clientes por DNI, nombre o categoría. Por otro lado, los asistentes a eventos se registran por su DNI y puede haber diferentes listas de eventos(VIP, Estudiante, Senior).

El programa debe proporcionar el siguiente menú de opciones:

- 1.Añadir/Modificar Cliente:** Permite ingresar o modificar la información de un cliente, incluyendo DNI, nombre, teléfono y categoría.
- 2.Buscar Cliente:** Permite buscar a un cliente por DNI, nombre o categoría, mostrando la información asociada.
- 3.Crear Evento:** Permite crear una conjunto de asisitentes a un evento por categoría. Por ejemplo: Nochevieja_VIP, Nochevieja_Estudiantes, etc.
- 4.Añadir Asistente a Evento:** Permite ingresar a un asistente a un evento indicando el nombre del evento y su DNI.
- 5.Listar Eventos:** Muestra una lista de todos los eventos registrados.
- 6.Salir:** Termina la ejecución del programa.

Implementa el programa utilizando diccionarios para almacenar información de clientes y conjuntos (sets) para gestionar los asistentes a eventos y los eventos en diferentes categorías.

Programación estructurada

Con la **programación modular y estructura**, 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 algoritmo (divide y vencerás). Además el uso de subrutinas nos proporciona la **reutilización del código** y no tener **repetido instrucciones** que realizan la misma tarea.

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

```
>>>  
factorial( 5)  
120
```

Para **llamar a una función** se debe utilizar su nombre y entre paréntesis los **parámetros reales** que se mandan. La llamada a una función se puede considerar una expresión cuyo valor y **tipo** es el retornado por la función.

Definición de funciones

```
def nombre(parametro1, parametro2):  
    print parametro1  
    print parametro2  
    return parametro1 + parametro2
```

Es decir, la palabra clave **def** seguida del nombre de la función y entre paréntesis los argumentos separados por comas.

En otra línea, indentado y después de los dos puntos tendríamos las líneas de código que conforman el código a ejecutar por la función.

Funciones

docstrings

```
def suma(parametro1, parametro2):  
    """Calcula la suma de los dos  
    numeros"""  
    return parametro1 + parametro2
```

La primera línea de la función debería ser un comentario (triple entrecomillado) que describe clara y concisamente (una única línea, si es posible) el propósito de la función.

Ámbito de variables

Variables locales: se declaran dentro de una función y no se pueden utilizar fuera de esa función

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

Variables globales: son visibles en todo el módulo.

```
>>> PI = 3.1415  
>>> def area(radio):  
...     return PI*radio**2  
...  
>>> area(2)  
12.566
```

Parámetros formales y reales

```
def CalcularMaximo(num1,num2):  
    if num1 > num2:  
        return num1  
    else:  
        return num2
```

Parámetros formales: Son las variables que recibe la función, se crean al definir la función.

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

```
numero1 = int(input("Dime el número1:"))  
numero2 = int(input("Dime el número2:"))  
num_maximo = CalcularMaximo(numero1,numero2)  
print("El máximo es ",num_maximo;)
```

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 objetos y referencias.

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]
```

Devolución de información

Una función en python puede devolver información utilizando la instrucción **return**. La instrucción **return** puede devolver cualquier tipo de resultados, por lo tanto **es fácil devolver múltiples datos guardados en una lista, tupla o diccionario**.

Aunque podemos cambiar el parámetro real cuando los objetos pasados son de tipo mutables, **no es recomendable hacerlo en Python**.

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'>
```

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(num):  
    if num==0 or num==1:  
        return 1  
    else:  
        return num * factorial(num-1)
```


Funciones

Parámetros por defecto

```
def bienvenida(nombre, saludo="Hola"):  
    print saludo + ", " + nombre
```

El valor por defecto es utilizado si ningún valor es indicado para ese parámetro al llamar la función.

Estos parámetros han de ir siempre al final de la lista de argumentos de la función.

```
>>> bienvenida("Sara")  
Hola, Sara  
  
>>> bienvenida("Baby", "Sayonara")  
Sayonara, Baby  
  
>>> bienvenida("Cesar", saludo="Ave")  
Ave, Cesar
```

Funciones

return

return

Sale de la función, devolviendo el valor indicado.

En Python las funciones siempre devuelven un valor; si no indicamos ninguno, el intérprete devolverá None (similar al NULL)

Es posible devolver mas de un valor a la vez.

Python devuelve los valores encapsulados en una *tupla* cuyos elementos son los valores a retornar.

```
>>> def max_y_min(x, y):  
...     if x > y:  
...         return x, y  
...     else:  
...         return y, x  
  
>>> max_y_min(4, 2)  
(4, 2)  
  
>>> max_y_min(6, 9)  
(9, 6)
```

Funciones

Expansión de parámetros

```
>>> mayor, menor = max_y_min(-4, 11)
>>> mayor
11
>>> menor
-4
```

Hay que asignar valores a dos variables y la función devuelve dos elementos: Python asigna el primer valor a la primera variable y el segundo a la segunda.

Funciones

Duck typing

```
int suma(int x, int y) {  
    return x + y;  
}
```

"Cuando veo un ave que camina como un pato, nada como un pato y suena como un pato, a esa ave yo la llamo un pato"
-- James Whitcomb Riley

La función comprueba (exige) que tanto x como y sean enteros

```
def suma(x, y):  
    """Calcula la suma de los dos numeros"""  
    return x + y
```

Directamente suma ambos valores, sin importar qué sean.

Funciones

Duck typing

Es el conjunto de métodos y propiedades lo que determina la validez semántica, en lugar del tipo del dato.

Nuestra función recibe dos variables y las suma.

No es necesario comprobar si son enteros, flotantes, complejos o cualquier otro tipo de dato que pueda sumarse.

Es mas fácil pedir perdón que permiso.

No se comprueba el tipo de dato recibido; en su lugar, se confía en la buena documentación y se actúa solo en caso de error.

Funciones lambda

lambda parámetros: expresión

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

funciones anónimas porque se definen sin un nombre.

```
>>> def cuadrado(x):  
...     return x ** 2
```

```
>>> cuad = lambda x: x ** 2  
>>> print(cuadrado(3))  
9  
>>> print(cuad(5))  
25
```

```
>>> def f(x, y, z=1):  
...     return (x+y) * z  
>>> f(5, 6)  
11  
>>> f(5, 6, 7)  
77  
>>> f = lambda x, y, z=1: (x+y) * z  
>>> f(5, 6)  
11  
>>> f(5, 6, 7)  
77
```

Proceso avanzado de secuencias

```
a = [1, 5, 3, 7, 6, 3, 2, 4]
```

¿Cómo extraer los números pares de la lista?

```
>>> b = []
```

```
>>> for numero in a:
```

```
...     if numero % 2 == 0:
```

```
...         b.append(numero)
```

Proceso avanzado de secuencias

filter

filter(funcion, secuencia)

Recibe dos argumentos: una función y una secuencia, devolviendo otra lista con los elementos de la secuencia para los cuales la función devuelve True.

En otras palabras, `filter` devuelve una lista con los elementos de la secuencia que satisfacen cierta condición.

```
>>> lista = [1, 5, 3, 7, 6, 3, 2, 4]
>>> filter(lambda x: x % 2 == 0, lista)
[6, 2, 4]
```

Ejecuta la función lambda para cada elemento de la lista, seleccionando únicamente aquellos para los cuales esta devolvió True (i.e., los pares)

Proceso avanzado de secuencias

map

```
a = [1, 5, 3, 7, 6, 3, 2, 4]
```

¿Cómo calcular el cuadrado de cada elemento?

```
>>> b = []
```

```
>>> for numero in a:
```

```
...     b.append(numero ** 2)
```

Proceso avanzado de secuencias

map

map(funcion, secuencia)

Recibe dos argumentos: una función y una secuencia, devolviendo otra lista con el resultado de aplicar `funcion` a cada elemento.

Es decir, devuelve el resultado de ejecutar `funcion` para cada uno de los elementos de secuencia.

```
>>> lista = [1, 5, 3, 7, 6, 3, 2, 4]
>>> map(lambda x: x ** 2, lista)
[1, 25, 9, 49, 36, 9, 4, 16]
```

Ejecuta la función `lambda` para cada elemento de la lista.

Listas por comprensión

La máxima expresión del poder de Python.

```
a = [1, 5, 3, 7, 6, 3, 2, 4]
b = [x ** 2 for x in a if x % 2 == 0]
```

Crea una lista. . .

. . . tomando el cuadrado. . .

. . . de cada uno de los elementos de a. . .

. . . que son pares.

Listas por comprensión

```
>>> a = [1, 5, 3, 7, 6, 3, 2, 4]
>>> [x ** 2 for x in a]
[1, 25, 9, 49, 36, 9, 4, 16]
```

Así elevamos al cuadrado todos los números. . .

```
>>> a = [1, 5, 3, 7, 6, 3, 2, 4]
>>> [x ** 2 for x in a if x % 2 == 0]
[36, 4, 16]
```

. . . y así sólo de aquellos que son pares.

Excepciones

Excepciones

Errores en tiempo de ejecución.

Python "lanza" la excepción cuando encuentra el error.

El programa finaliza inmediatamente.

La alternativa sería continuar un proceso que no tiene sentido.

Errores sintácticos

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

Errores en tiempo de ejecución (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
```

Excepciones

Las excepciones también indican dónde (fichero y línea) se ha producido la excepción, facilitándonos la depuración del programa.

```
Traceback (most recent call last):
```

```
File prueba.py, line 2, in <module>
```

```
    5 / 0
```

```
ZeroDivisionError: integer division or modulo by  
zero
```

Hemos dividido por cero en la segunda línea del fichero `prueba.py`

Excepciones

`NameError`: Referencia a una variable no declarada

```
>>> a = 1
```

```
>>> a += b
```

```
NameError: name 'b' is not defined
```

`ValueError`: Buscar en una lista un valor que no existe

```
>>> a = [1, 2, 3]
```

```
>>> a.index(5)
```

```
ValueError: list.index(x): x not in  
list
```

Excepciones

`TypeError`: Mezclar tipos de datos sin convertirlos previamente

```
>>> 1 + "2"
```

`TypeError`: unsupported operand type(s)

for +: 'int' and 'str'

`IOError`: Errores en entrada y salida de datos

```
>>> fd = open("inexistente", "r")
```

`IOError`: [Errno 2] No such file or
directory: 'inexistente'

Listado completo: <http://docs.python.org/library/exceptions.html>

Manejando excepciones

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

```
>>> 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")
```

Manejando excepciones

```
>>> 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")
```

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

try...except

Manejo de excepciones

```
try:
    fd = open("inexistente")
except IOError:
    print "El fichero no existe!"
```

except

Permite seleccionar qué excepciones se capturan y cómo procesarlas.

```
except IOError:
    print "El fichero no existe!"
```

Excepciones múltiples

```
try:
    fd = open("fichero")
    a = 3 / 0
except IOError:
    print "El fichero no
existe"
except ZeroDivisionError:
    print "No se puede
dividir"
```

¿Y para manejar de forma idéntica
ambas excepciones?

```
try:
    fd = open("fichero")
    a = 3 / 0
except (IOError, ZeroDivisionError):
    print "Error en la fase de
inicialización."
```

- **except se ejecutará tanto si IOError como ZeroDivisionError son lanzadas.**

Introducción

¿Qué son?

Un módulo es un fichero que contiene código PYTHON.

Su extensión es `.py`.

Almacena declaración de **variables** e implementación de **funciones**.

Posibilidad de hacer **referencia a otros modulos** (mediante la instrucción `import`).

La función principal es **organizar** el código.

Como resultado:

1. Limpieza y orden
2. Será mucho más sencillo **reutilizar** nuestro código.

Podremos publicar nuestro código para que lo use cualquiera

Introducción a los módulos

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.

Importación de módulos

```
>>> import math
>>> math.sqrt(9) 3.0
>>> from math import sqrt
>>> sqrt(9) 3.0
```

Módulos de hora y fecha

- **time**
- ```
>>> time.sleep(1)
```
- **datetime**

## Módulos matemáticos

- **math**
- ```
>>> math.sqrt(9)
```
- **functions**
- **random**
- ```
>>> random.randint(1,10)
```

## Módulos del sistemas

- **os**
- ```
>>> os.system("clear")
```

Introducción

Ejemplo de módulo (areas.py)

```
"""Modulo para el calculo de areas de formas basicas"""  
pi = 3.1416  
  
def cuadrado(lado):  
    """Calcula el area del cuadrado a partir de su lado"""  
    return lado ** 2  
  
def circulo(radio):  
    """Calcula el area del circulo dado el radio"""  
    return pi * radio ** 2  
  
print ('Area cuadrado =', cuadrado(2))  
print ('Area circulo =', circulo(1))
```

Introducción

¿Cómo se usan?

Pueden ser ejecutados desde consola

```
python areas.py
```

Pueden ser "llamados" por otros módulos o desde la consola de PYTHON, mediante la instrucción

```
import areas
```

En la importación, se escribe el nombre del fichero sin el .py.

Se obtiene información sobre el contenido del módulo mediante la instrucción

```
help(areas)
```

Se accede a su contenido con la sintaxis

```
nombreModulo.variable o nombreModulo.funcion
```

```
print areas.pi;
```

```
print areas.cuadrado(3)
```

¿Cómo se usan los módulos?

Otras instrucciones de importación (I)

Podemos importar todo el módulo a nuestro espacio de trabajo con la instrucción

- `from nombre_modulo import *`
- Ya no es necesario escribir `nombre_modulo.funcion([parametros])`.
- Ahora basta con `funcion([parametros])`. Aunque no se recomienda esta práctica al escribir nuevos módulos, es muy útil cuando estamos haciendo pruebas en el intérprete.

Puedo importar sólo ciertas partes de módulo (ya sean funciones o variables).

- `from nombre_modulo import variable`
- `from nombre_modulo import funcion`
- De nuevo, puedo hacer uso de lo importado simplemente escribiendo `variable` o `funcion([parametros])` donde sea necesario en mi código.

¿Cómo se usan los módulos?

Otras instrucciones de importación (II)

Puedo generar un alias para un módulo (si el nombre es demasiado largo o difícil de escribir)

- `from nombre_modulo_difícil_y_largo as modalias`
- Ahora puedo acceder al contenido del módulo como `modalias.variable` o `modalias.funcion([parametros])`.

Finalmente, puedo poner alias a variables o funciones del modulo

- `from nombre_modulo_difícil import variable as varalias`
- Podré acceder a ese contenido escribiendo sólo el alias `varalias`.

¿Cómo se usan los módulos?

Otras instrucciones de importación: Ejemplo

```
"""Modulo para el calculo de areas de formas basicas"""  
pi = 3.1416  
  
def cuadrado(lado):  
    """Calcula el area del cuadrado a partir de su lado"""  
    return lado ** 2  
  
def circulo(radio):  
    """Calcula el area del circulo dado el radio"""  
    return pi * radio ** 2  
  
print 'Area cuadrado =', cuadrado(2)  
print 'Area circulo =', circulo(1)
```

¿Cómo se usan los módulos?

Otras instrucciones de importación: Ejemplo

```
>>> import areas
>>> areas.pi
3.1416
>>> areas.cuadrado(3)
9
>>> from areas import *
>>> circulo(1)
3.1416
>>> pi
3.1416
```

```
>>> from areas import cuadrado
>>> cuadrado(4)
16
>>> import areas as a
>>> a.pi
3.1416
>>> from areas import circulo as ac
>>> ac(2)
12.5664
```

Ejercicios para practicar

1. Cree un módulo llamado `operaciones_basicas.txt` con las siguientes funciones: sumar, restar, multiplicar, dividir.
Cada función recibirá dos parámetros (los operandos). Cada función devolverá el resultado de la operación. Controle la excepción de la división por cero en la función dividir.
Documente el módulo y las funciones con las siguientes cadenas de texto:
Modulo --> Módulo Operaciones_basicas.
sumar --> Función que suma los dos parametros y devuelve el resultado.
restar --> Función que resta los dos parametros y devuelve el resultado.
multiplicar --> Función que multiplica los parámetros y devuelve el resultado.
dividir --> Función que divide los parámetros (numerador, denominador) y devuelve el resultado. Lanza una excepción en caso de división por cero. El código de esa excepción mostrará el mensaje *'ERROR: No se puede dividir por cero'* y lanzará una excepción del tipo *ZeroDivisionError*.

Ejercicio

Importa el módulo del ejercicio 22 en el siguiente código, de forma que las instrucciones siguientes se ejecuten sin producir errores:

```
a, b = 13, 3

print 'Operandos =', a, b

print 'sumar = ', operaciones_basicas.sumar (a, b)

print 'restar = ', operaciones_basicas.restar (a, b)

print 'multiplicar = ', operaciones_basicas.multiplicar (a, b)

print 'dividir = ', operaciones_basicas.dividir (a, b)
```

¿Cómo debería importar el módulo para que no hubiera errores al ejecutar?

```
a, b = 13, 3

print 'Operandos =', a, b

print 'sumar = ', ob.sumar (a, b)

print 'restar = ', ob.restar (a, b)

print 'multiplicar = ', ob.multiplicar (a, b)

print 'dividir = ', ob.dividir (a, b)
```

¿Y cómo importaría para que funcionara el siguiente código?

```
a, b = 13, 0

print 'Operandos =', a, b

print 'sumar = ', sumar (a, b)

print 'restar = ', restar (a, b)
```

**Curso Introducción a la
programación con Python3**

**Programación
orientada a objetos**

Introducción a la Programación orientada a Objetos

```
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 mostrar(self):
        return str(self.x)+":"+str(self.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))
```

Llamamos **clase** a la representación abstracta de un concepto. Las clases se componen de **atributos** y **métodos**.

Los **atributos** definen las características propias del objeto y modifican su estado.

Los **métodos** son bloques de código (o funciones) de una clase que se utilizan para definir el comportamiento de los objetos.

El **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 (objeto).

Todos los métodos de cualquier clase, recibe como primer parámetro a la instancia sobre la que está trabajando. (**self**).

Introducción a la Programación orientada a Objetos

```
>>> punto1=punto()  
>>> punto2=punto(4,5)  
>>> print(punto1.distancia(punto2))  
6.4031242374328485
```

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

```
>>> punto2.x  
4  
>>> punto2.x = 7  
>>> punto2.x  
7
```

Podemos acceder y modificar los atributos de objeto.

Encapsulamiento

La característica de no acceder o modificar los valores de los atributos directamente y utilizar métodos para ello lo llamamos **encapsulamiento**.

```
>>> 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'
```

Las variables que comienzan por un doble guión bajo __ la podemos considerar como **atributos privados**.

Encapsulamiento

- En Python, las **propiedades (getters)** nos permiten implementar la funcionalidad exponiendo estos métodos como atributos.
- Los métodos **setters** son métodos que nos permiten modificar los atributos a través de un método.

```
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
Radio debe ser positivo
>>> c1.radio
0
```

Herencia

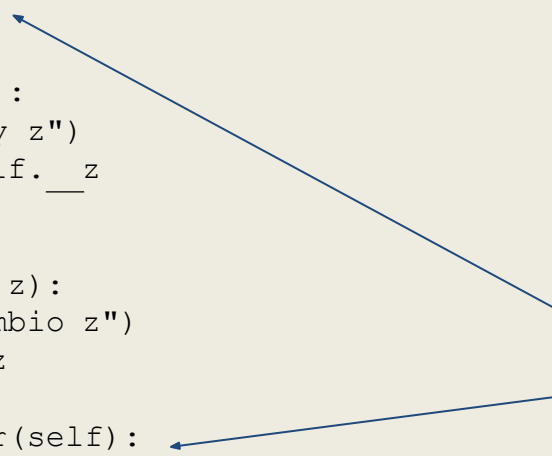
La clase **punto3d** hereda de la clase punto todos sus propiedades y sus métodos. En la clase hija hemos añadido la propiedad y el setter para el nuevo atributo z, y hemos modificado el constructor (sobrescritura) el método mostrar y el método distancia.

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

    @z.setter
    def z(self, z):
        print("Cambio z")
        self.__z=z

    def mostrar(self):
        return super().mostrar()+":"+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
```



```
>>> p3d=punto3d(1,2,3)
>>> p3d.x
1
>>> p3d.z
3
>>> p3d.mostrar()
1:2:3
>>> p3d.y = 3
```

La función **super()** me proporciona una referencia a la clase base. Nos permite acceder a los métodos de la clase madre.

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.

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

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

```
>>> c1=circulo(punto(2,3),5)
>>> print(c1.mostrar())
Centro:2:3-Radio:5
>>> c1.centro.x = 3
```


Libreria estandar de Python

¿Qué es?

La librería estándar contiene varios tipos de componentes.

Tipos de datos que pueden ser considerados como parte del núcleo de Python (números y listas).

Funciones internas y excepciones.

La mayor parte de la librería esta constituida por un amplio conjunto de módulos que permiten expandir la funcionalidad de Python.

Se distribuye junto con el intérprete.

Con cada nueva versión de Python, se mejora y amplía la funcionalidad.

<http://docs.python.org/library/index.html>

USO DEL SISTEMA OPERATIVO

Módulo sys

Ya son conocidas sus capacidades para la gestión de parámetros de entrada a scripts Python

- *propiedad argv*

También se ha utilizado para modificar la ruta de búsqueda de módulos Python

- *propiedad path*

USO DEL SISTEMA OPERATIVO

Módulo sys: Forzar fin de ejecución

Aunque no es programación de estilo, supongamos que, en un momento dado, bien porque el script tarda demasiado en ejecutarse, o porque se da cierta condición crítica, necesitamos terminar la ejecución de un script.

```
sys.exit([status])
```

donde `status` es un entero que sirve para informar del motivo de la finalización.

USO DEL SISTEMA OPERATIVO

Módulo sys: Forzar fin de ejecución. Ejemplo

Fichero: salida.py

```
import sys
if len(sys.argv) < 2:
    sys.exit(1)
print sys.argv[1:]
```

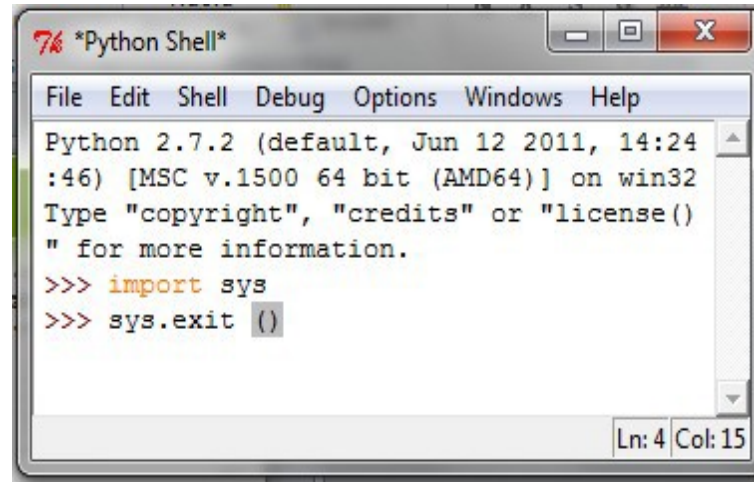
Desde el intérprete

```
>>> import sys
>>> import subprocess
>>> subprocess.call(['py
thon', 'salida.py'])
1
>>> subprocess.call(['py
thon', 'salida.py', 'par
ametro'])
['parametro']
0
```

USO DEL SISTEMA OPERATIVO

Módulo sys: Forzar fin de ejecución. Ejemplo (II)

También sirve para salir del intérprete



```
*Python Shell*
File Edit Shell Debug Options Windows Help
Python 2.7.2 (default, Jun 12 2011, 14:24:46) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import sys
>>> sys.exit()
```

Ln: 4 Col: 15

TRABAJO CON FECHAS Y HORAS

Módulo datetime

- ☐ En ocasiones necesito saber la hora a la que se ejecutó un script y/o el tiempo transcurrido.
- ☐ A veces necesito saber el intervalo temporal entre dos fechas.
- ☐ Quizá necesite parar la ejecución de un programa tras un intervalo de tiempo de espera (una petición web sin respuesta).
- ☐ Todo esto y más se puede realizar con el módulo **datetime**

TRABAJO CON FECHAS Y HORAS

Módulo datetime

Atributos

Dato tipo datetime.datetime

- year, month, day, hour, minute, second, microsecond

Dato tipo datetime.time

- hour, minute, second, microsecond

TRABAJO CON FECHAS Y HORAS

Módulo datetime

Creación alternativa

Podemos crear objetos tipo `datetime.time` y

`datetime.date` a partir de objetos `datetime.datetime`

```
>>> import datetime
```

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

```
>>> d = now.date()
```

```
>>> t = now.time()
```

TRABAJO CON FECHAS Y HORAS

Módulo datetime

Operaciones

Podemos realizar operaciones con fechas.

Operación	Operador	Ejemplo
Suma	+	$t1 + t2$
Resta	-	$t1 - t2$
Comparaciones	$==, !=, <, <=, >, >=$	$t1 < t2$

El resultado es un objeto del tipo `timedelta`.

Atributos de `timedelta`: `days`, `seconds` y `microseconds`.

Métodos de `timedelta`: `total_seconds()`

Entonces, estaríamos en condiciones de determinar, por ejemplo, la duración de nuestros programas o de fragmentos de código que sospechamos que ralentizan la ejecución.

TRABAJO CON FECHAS Y HORAS

Módulo datetime

Ejemplo

Tiempo que tardo en ejecutar unas instrucciones

```
>>> import datetime
>>> import time
>>> date1 = datetime.datetime.now()
# Esperamos unos segundos...
>>> date2 = datetime.datetime.now()
>>> datediff = date2 - date1
>>> datediff.days, datediff.seconds,
datediff.microseconds
(0, 15, 829942)
```

Ejercicio 28

¿Cuántos días faltan para su cumpleaños?

Si la clase termina a las 14:30 horas, ¿cuánto tiempo le queda para terminar la clase de hoy?

FICHEROS Y DIRECTORIOS

Módulos `os` y `shutil`

¿Cómo obtengo el directorio en el que estoy ejecutando órdenes en un script o intérprete?

¿Cómo creo y borro directorios y ficheros?

¿Cómo accedo al contenido de un directorio?

¿Cómo copio, muevo o borro un fichero? ¿Cómo realizo esas operaciones de forma recursiva sobre un directorio?

Todo esto y más con los módulos : **`os`** y **`shutil`**

FICHEROS Y DIRECTORIOS

Módulos os y shutil

Directorio de trabajo

```
>>> import os
>>> os.getcwd()
'E:\\presentaciones\\pyhon_inicial\\codigo'
>>> os.getcwd()
'C:\\'
```

FICHEROS Y DIRECTORIOS

Módulos os y shutil

Directorios: Crear

Ejemplo:

```
>>> os.mkdir ('pruebaDir')
```

```
>>> os.mkdir('pruebaDir\\subdir1\\subdir1_1')
```

```
Traceback (most recent call last):
```

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

```
OSError: [Errno 2] No such file or directory:
```

```
'pruebaDir\\subdir1\\subdir1_1'
```

```
>>> os.makedirs('pruebaDir\\subdir1\\subdir1_1')
```