DAM/DAW

PROGRAMACIÓN

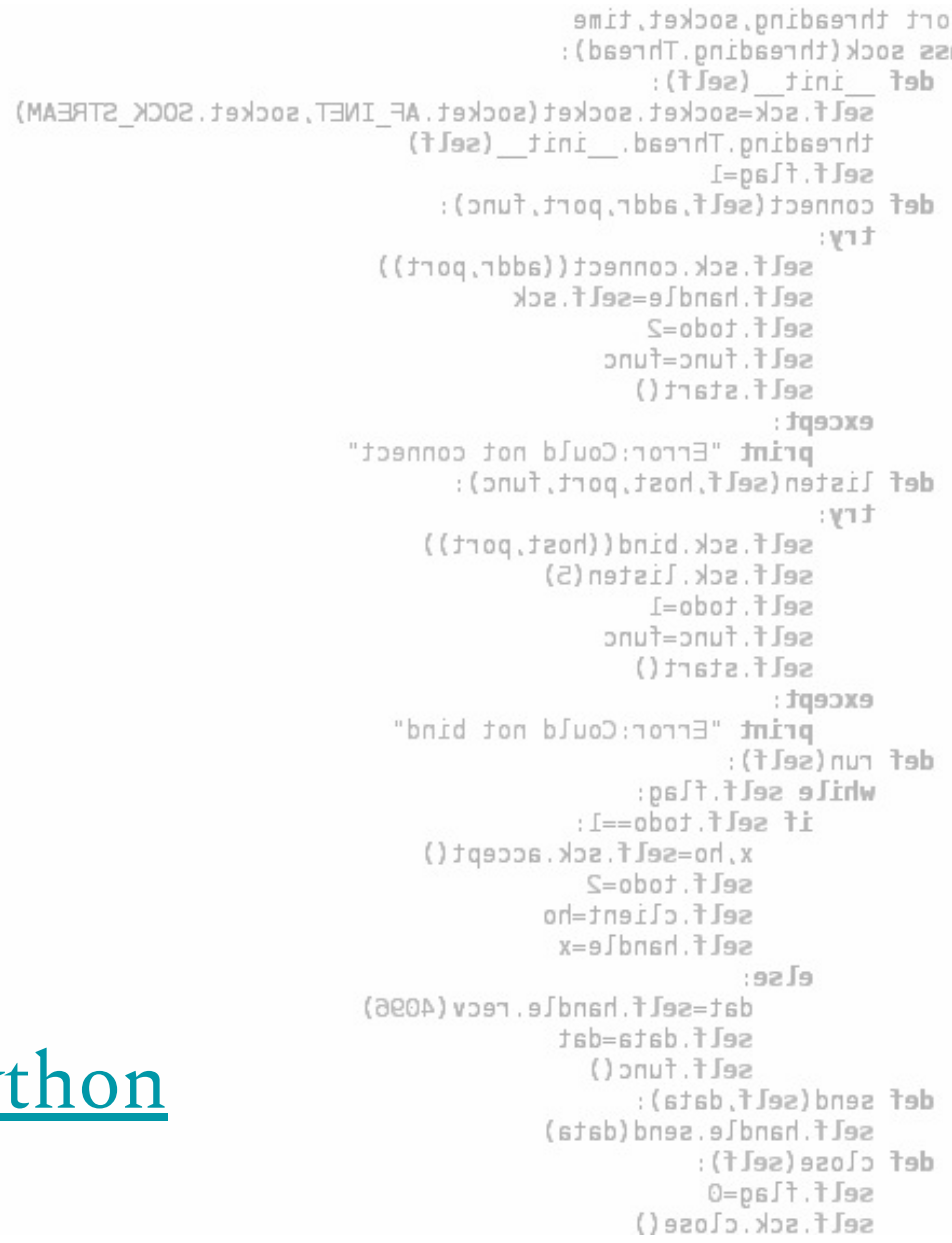
04.1 - Python

Estructuras de control II

Rev: 3.1

fran@iessanclemente.net

Indice



Introducción (I)

Funciones

- En muchos de los ejemplos visto hasta ahora, ya hemos hecho uso de funciones proporcionadas por el propio lenguaje: *print()*, *input()*, *sqrt()*,... Estas estructuras del lenguaje son la base de la programación modular, que se cimenta sobre dos conceptos clave: abstracción y descomposición

- En cuanto a la abstracción, las funciones actúan como cajas negras a las que les proporcionamos unos datos de entrada (argumentos, como la cadena de texto que le pasamos a *print()* para que la imprima), realizan un determinada acción y, finalmente, devuelven un valor de retorno (por ejemplo, los datos introducidos por el usuario que devuelve la función *input()*). Pero todos los detalles de su implementación, su funcionamiento interno, queda oculto

Introducción (y II)

- Desde el punto de vista de la descomposición, el empleo de funciones nos permite la división de problemas complejos en tareas más simples y abordables computacionalmente.
- En general, estas unidades funcionales:
  - son autocontenidas, es decir, representan tareas bien definidas que la función puede resolver (si dispone de los datos necesarios)
  - permiten la división del código, facilitando su organización y mantenimiento
  - son reutilizables
  - facilitan el trabajo en equipo ya que permiten repartir mejor la carga de trabajo entre diferentes programadores
- Permiten su inclusión en bibliotecas o librerías de forma que sólo se programen una vez pero se puedan usar por múltiples programas

Definición

- Python nos permite crear nuestras propias funciones. Se declaran con la cláusula def y todas tendrán:
  - un nombre
  - parámetros (0 o más), encerrados entre paréntesis ( )
  - un *docstring* (opcional), con la descripción de la función, parámetros y retorno
  - un cuerpo (acciones)
  - cláusula return (0 o más) para devolver un valor. Si no existe, devuelve *None*

```
def es_par(i):
    """Chequea si un número es par.

    Args:
        i (int): número entero.
    Returns:
        (bool): True si es par.
    """
    if i%2 == 0:
        par = True
    else:
        par = False

    return par
```

Invocación

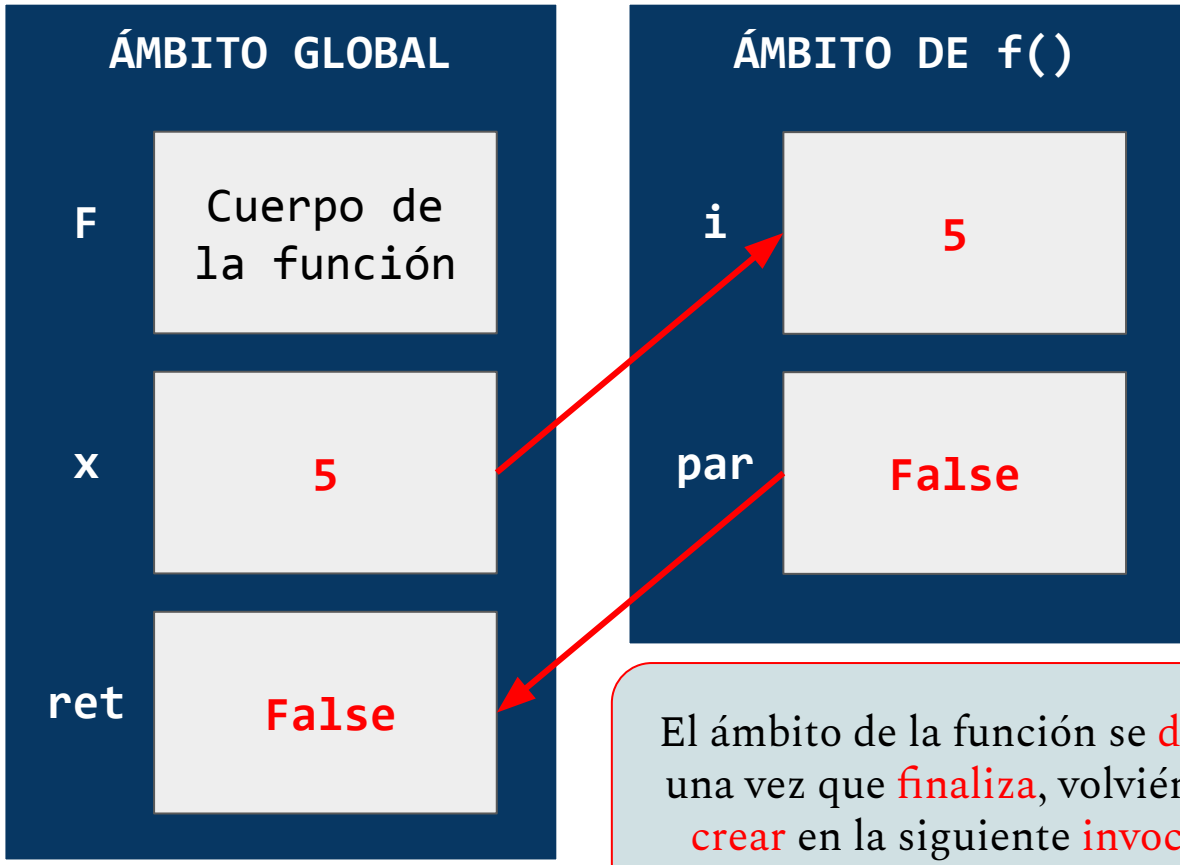
- El código de una función, es decir, su cuerpo, sólo se ejecutará cuando la llamemos (invocar) desde algún punto de nuestro programa
- La función deberá definirse antes de que pueda ser invocada

```
def f(i):
    if i%2 == 0:
        par = True
    else:
        par = False

    return par

# Inicio del programa
x = int(input('Número?'))
ret = f(x) # invocación
if ret:
    print('Par')
else:
    print('Impar')
```

cuerpo



El ámbito de la función se destruye una vez que finaliza, volviéndose a crear en la siguiente invocación

Retorno (I)

- La cláusula return se emplea para devolver el control del programa al punto desde el que se llamó a la función (*invocación*), y devolver un valor de retorno que podrá ser empleado en cualquier expresión
- Una función puede tener varias cláusulas return (sólo una se ejecutará) y el valor devuelto puede ser resultado de cualquier expresión, incluso llamadas a otras funciones. Las siguiente funciones son equivalentes:

```
def es_par(i):
    if i%2 == 0:
        return True
    else:
        return False
```

```
def es_par(i):
    return i%2 == 0
```

- En Python, todas las funciones devuelven un valor. En el caso de que nuestra función no tenga cláusula return, se devolverá de forma implícita el valor *None*

Retorno (II)

- Intenta determinar la salida del siguiente programa:

```
def func_a():
    print("dentro de func_a")

def func_b(y):
    print("dentro de func_b")
    return y

def func_c(z):
    print("dentro de func_c")
    return z()

print(func_a())
print(5 + func_b(2))
print(func_c(func_a))
```

```
dentro de func_a
None
dentro de func_b
7
dentro de func_c
dentro de func_a
None
```

Retorno (y III)

- Las funciones en Python, pueden devolver más de un valor:

```
def min_max(vals):
    """Devuelve el máximo y el mínimo de una lista de valores.

    Args:
        vals (List): lista de valores.
    Returns:
        (float, float): Mínimo y máximo.
    """
    vals.sort()
    return vals[0], vals[-1]

valores = [7, 9, 2.5, 3, 12.2]
min, max = min_max(valores)
print("Mínimo:", min, "; Máximo:", max)
```

return devuelve varios valores separados por comas

recogemos los valores devueltos en varias variables separadas por comas. No es obligatorio recoger todos los valores devueltos

```
Mínimo: 2.5 ; Máximo: 12.2
```

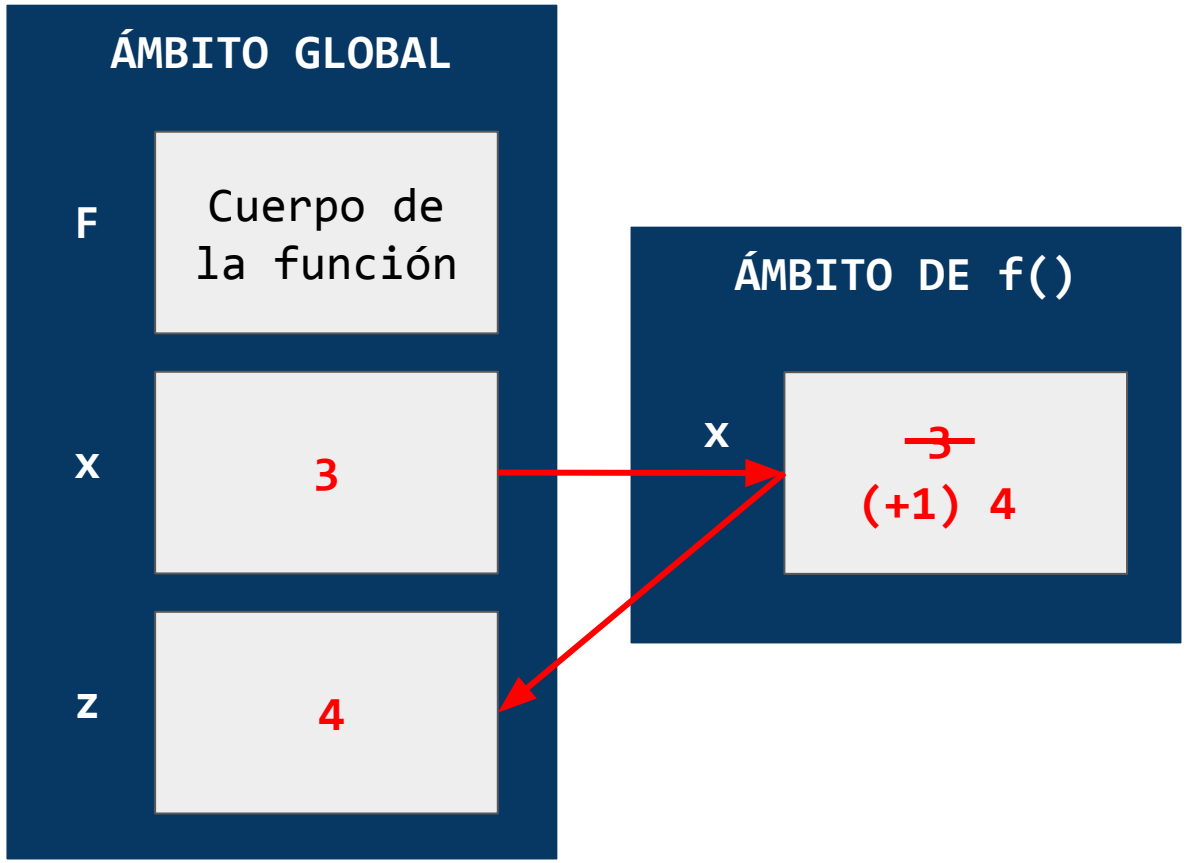
Ámbito de variables (I)

- Es importante comprender que la/las variable/s argumento de la definición de la función (*formal parameters*), y que sólo son visibles en el cuerpo de la misma, son independientes de las variables empleadas como argumento en la invocación (*actual parameters*)

```
def f( x ):      Formal parameter
    x = x + 1
    print("en f(x): x =", x)
    return x

x = 3
z = f( x )       Actual parameter
print("x =",x,"; z =",z)
```

```
en f(x): x = 4
x = 3 ; z = 4
```



Ámbito de variables (II)

- Una característica de Python es que, dentro de la función, se puede acceder a las variables definidas fuera, pues se consideran globales, pero, en principio, no se pueden modificar.

```
def f(x):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

x local

```
2
5
```

```
def g(y):
    print(x)
    print(x+1)

x = 5
g(x)
print(x)
```

x global

```
5
6
5
```

```
def h(y):
    x = x+1

x = 5
h(x)
print(x)
```

```
UnboundLocalError:
local variable 'x'
referenced before
assignment
```

Ámbito de variables (III)

- Cuando queremos acceder a variables externas de ámbito global desde una función para modificarlas, Python nos permite declararlas dentro de la función mediante la cláusula global

```
def h():
    global x
    x = x+1

x = 5
h()
print(x)
```

x global

```
6
```

- Aunque el lenguaje lo permite, no debería hacerse uso en general de esta facilidad, pues limitamos la independencia de la función (*autocontención*). Idealmente, todos los datos deberían llegarle a la función mediante sus parámetros

Ámbito de variables (y IV)

- En las llamadas a las funciones en Python, los *parámetros formales* de la función se inicializan con referencias a los objetos apuntados por los *parámetros actuales* de la llamada.
- Esto supone que, cuando estos objetos son de tipos de datos mutables (*listas*, *sets* y *diccionarios*), podremos realizar modificaciones desde dentro de la función a los objetos referenciados por los parámetros.

```
def min_max(vals):
    vals.sort()
    return vals[0], vals[-1]

valores = [7, 9, 2.5, 3, 12.2]
print("Antes:",valores)
min, max = min_max(valores)
print("Mínimo:", min, "; Máximo:", max)
print("Después:"valores)
```

lista

```
Antes: [7, 9, 2.5, 3, 12.2]
Mínimo: 2.5 ; Máximo: 12.2
Después: [2.5, 3, 7, 9, 12.2]
```

Argumentos (I)

- Nuestras funciones pueden tener el número de parámetros formales que precisemos. En la invocación de la función debemos proporcionar suficientes valores para todos ellos y en el orden indicado, según la especificación de la función

```
def autor(nom, apel, f_nac):
. . .
autor("Frank", "Miller", "27/01/1957")
```

Especificación

- Podemos alterar el orden de los argumentos en la invocación de la función, si empleamos parejas *nombre=valor* (para todos los parámetros)

```
def autor(nom, apel, f_nac):
. . .
autor(f_nac="27/01/1957", apel="Miller", nom="Frank")
```

Argumentos (II)

- Podemos establecer valores por defecto para los parámetros de la función. Para ello, se les asignará un valor en la definición y podrán ser omitidos en la invocación (se usarán los valores por defecto). Si hay parámetros obligatorios y opcionales, estos deben ir al final

```
def autor(nom, apel, f_nac, activo="s"):
. . .
autor("Frank", "Miller", "27/01/1957")
autor("Stan", "Lee", "28/12/1922", "n")
```

se usa el valor por defecto "s" de activo

se usa el valor "n" proporcionado

- Muchas de las funciones de la librería estándar que usamos habitualmente tienen parámetros con valores por defecto

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
open(file, mode='r', buffering=-1, encoding=None, errors=None,
    newline=None, closefd=True, opener=None)
```

Argumentos (III)

- Python nos permite definir funciones que acepten un número variable de argumentos. Para ello, usaremos los parámetros especiales **args* y ***kwargs* (args y kwargs pueden ser nombre arbitrarios).

*args

- Se creará en la función una tupla de nombre *args* y longitud variable que recogerá los parámetros pasados. Los argumentos pasados en la llamada pueden ser de diferente tipo

```
def sumatorio(*nums):
    print("sumar:",nums)
    suma = 0
    for n in nums:
        suma += n
    return suma
print(sumatorio(1, 2, 3))
print(sumatorio(15, 10, -10, 5))
```

```
sumar: (1, 2, 3)
6
sumar: (15, 10, -10, 5)
20
```

Argumentos (y IV)

**kwargs (*key-word arguments*)

- En este caso, se creará un diccionario de nombre *kwargs* y longitud variable que recogerá los parámetros pasados como parejas *nombre=valor*. Los argumentos pasados en la llamada pueden ser de cualquier tipo

```
def perso(**data):
    print("\n**data = ", end="")
    print(data)
    print()
    for k in data:
        print(k,">",data[k])


perso(nom="Frank", apel="Miller")
perso(nom apel="Taiyo Matsumoto",
edad=51, pais="Japón")
```

```
**data = {'nom': 'Frank', 'apel': 'Miller'}

nom > Frank
apel > Miller

**data = {'edad': 51, 'pais': 'Japón',
'nom_apel': 'Taiyo Matsumoto'}

edad > 51
pais > Japón
nom_apel > Taiyo Matsumoto
```

Módulos (I)

- A medida que nuestros programas van creciendo y se van haciendo más complejos, es necesario dividirlos de diferentes archivos para facilitar su mantenimiento
- De igual modo, es probable que algunas de las funciones que ya tenemos escritas, queramos reutilizarlas en algún otro programa sin tener que "copiar" en él la definición de las mismas
- Python nos permite escribir las definiciones de las funciones en archivos Python independientes o módulos.
- Para poder usar en un programa las funciones contenidas en un módulo, deberemos importar dicho módulo (o una función concreta) desde nuestro programa mediante el uso de la sentencia *import*
- El siguiente ejemplo, muestra un módulo (archivo ***fibo.py***) con dos funciones para resolver la serie de Fibonacci hasta un número dado

Módulos (II)

```
"""Módulo de la serie de Fibonacci (fibo.py)."""

def fib(n):
    """Imprime la serie de Fibonacci."""
    a, b = 0, 1     # Asignación simultánea de a y b
    while b < n:
        print (b, end=' ')
        a, b = b, a+b

def fib2(n):
    """Devuelve una lista con la serie de Fibonacci."""
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Módulos (III)

- Vamos a hacer un pequeño programa (***test_fibo.py***) para probar las funciones del módulo anterior (ambos archivos deben estar en la misma carpeta)

```
import fibo

num = int(input('Introduce un número entero: '))

# Imprime la serie de Fibonacci hasta num
fibo.fib(num)

# Obtiene una lista con la serie de Fibonacci hasta num
print(fibo.fib2(num))
```

importamos el módulo fibo

```
Introduce un número entero: 10
1 1 2 3 5 8
[1, 1, 2, 3, 5, 8]
```

Módulos (IV)

- La sentencia *import fibo* hace que el nombre del módulo se importe en la tabla de símbolos del programa, de forma que las distintas funciones que contiene pueden ser accedidas mediante la sintaxis:

  nombre_de_módulo.nombre_de_la_función

- Existe una variante de *import* que nos permite importar directamente los nombres de las funciones a la tabla de símbolos. De ese modo, podremos invocarlas directamente sin tener que anteponer el nombre del módulo:

  from nombre_de_módulo import nombre_de_la_función

- Aunque no es aconsejable, podemos usar:

  *from nombre_de_módulo import **

  para importar todos los nombres de funciones del módulo

Módulos (V)

- Nuestro anterior programa podría quedar:

```
from fibo import fib, fib2

num = int(input('Introduce un número entero: '))

# Imprime la serie de Fibonacci hasta num
fib(num)

# Obtiene una lista con la serie de Fibonacci hasta num
print(fib2(num))
```

importamos las unciones fib y fib2 del módulo fibo

invocamos la función sin utilizar el nombre del módulo

Módulos (VI)

alias de nombre de módulo/función

- La sintaxis de *import* permite la definición de *alias* de nombres

  import nombre_de_módulo as alias

  A partir de ese momento, deberemos invocar las funciones mediante:

  alias.nombre_de_la función

```
import fibo as f

num = int(input('Introduce un número entero: '))

# Imprime la serie de Fibonacci hasta num
f.fib(num)

# Obtiene una lista con la serie de Fibonacci hasta num
print(f.fib2(num))
```

creamos el *alias* f para el módulo fibo

usamos el *alias* f en lugar del nombre del módulo

Módulos (VII)

Búsqueda de módulos en el sistema

- Python busca nuestros módulos en la carpeta actual. Cuando están contenidos en diferentes subcarpetas, deberemos "replicar" esa misma estructura en la sentencia *import* concatenando las subcarpetas mediante puntos (.) Si en el ejemplo anterior, el archivo ***fibo.py*** del módulo estuviera en una subcarpeta denominada "**mods**" haríamos

  import mods.fibo (con *alias*, *import mods.fibo as f*)

  Las funciones se invocarían mediante:

  mods.fibo.fib y *mods.fibo.fib2* (con *alias*, *f.fib* y *f.fib2*, *no cambia!*)

- Podemos añadir directorios de búsqueda de módulos mediante la variable de entorno PYTHONPATH o, directamente en el programa:

```
import sys
sys.path.append('/ruta/al/directorio')
```

Módulos (y VIII)

La variable __name__

- Cuando el intérprete Python carga un nuevo archivo para su ejecución, inicializa la variable de sistema __name__ con el siguiente valor:
  - "__main__", si el archivo se lanzó como un *script* y es el punto de entrada de la aplicación (*main scope*)
  - *"nombre_del_modulo"*, si el archivo se importó como módulo
- Es habitual que los scripts de Python evalúen al iniciarse esta variable para determinar cómo fueron invocados y actuar en consecuencia. Por ejemplo, podríamos añadir lo siguiente a un módulo de funciones para ejecutar pruebas con doctest:

```
if __name__ = '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

La biblioteca estándar (I)

- Toda instalación de Python incluye la llamada Biblioteca Estándar de Python que, básicamente, es una colección de módulos (escritos en C o Python) que nos permiten añadir nuevas funcionalidades a nuestras aplicaciones.
- Entre otros, dispondremos de módulos orientados a la gestión de I/O en archivos, desarrollos en entornos gráficos, comunicaciones TCP/IP, operaciones matemáticas, compresión, multimedia, etc...
- El siguiente ejemplo, emplea una función del módulo *random* para generar números aleatorios dentro del rango establecido por dos valores límite

```
from random import randint


for i in range(0, 10):
    print(randint(0, 5), end=' ')
```

La biblioteca estándar (y II)

- Los siguientes ejemplos muestran cómo leer los parámetros pasados al programa desde la línea de comandos, o cómo crear un sistema de *log*

```
if __name__ = '__main__':
    import sys, logging

    logging.basicConfig(filename="log.txt", level=logging.DEBUG,
            format='%(asctime)s %(levelname)-8s %(message)s')
    logging.info('Lista de argumentos: ', ' '.join(sys.args))

    for arg in sys.argv:
        print(arg)
```

- La documentación oficial de la Librería Estándar de Python se encuentra en: https://docs.python.org/3/library/index.html
- La lista de módulos en: https://docs.python.org/3/py-modindex.html
- Otro recurso de interés: https://pymotw.com/2/index.html

PyPI. El repositorio de Python (I)

- Python dispone de un sistema de distribución de *software* basado en paquetes. Por ej., la *Python Standard Library* es un colección de paquetes
- El *Python Package Index* (PyPI) és un repositorio de *software* para Python que incluye en la actualidad más de 150 mil proyectos desarrollados por la comunidad.
- Los desarrolladores de Python pretenden que sea un catálogo exhaustivo de todos los paquetes de Python en código abierto
- PyPI por un lado facilita la búsqueda e instalación de paquetes de *software* para Python (herramientas, aplicaciones, librerías,...) y, por otro, sirve de portal para que los desarrolladores puedan publicar sus paquetes y distribuir su *software*
- El sitio principal de PyPI es: https://pypi.org/

PyPI. El repositorio de Python (II)

- Aunque podemos descargar directamente los paquetes de PyPI e instalarlos manualmente, lo habitual es emplear la herramienta *pip* (*pip3*) incluída con la distribución de Python
- *pip*, similar a la herramienta ***apt*** de Debian (Ubuntu), nos permite (des)instalar, buscar,... paquetes del repositorio. Si lo ejecutamos sin parámetros, nos mostrará una lista de sus principales opciones
- *pip* se puede usar como administrador, en cuyo caso los paquetes descargados quedarán disponible para todos los usuarios del sistema (*/usr/lib/python3/dist-packages*), o como usuario estándar (por ejemplo, se instalarán en *~/.local/lib/python3.6/site-packages*). En este caso, los paquetes sólo estarían disponibles para ese usuario
- *pip* instalará aquellos otros paquetes Python necesarios para satisfacer las dependencias del paquete que queremos instalar

PyPI. El repositorio de Python (III)

- A modo de ejemplo, vamos a instalar un paquete del repositorio denominado *matplotlib*. Este paquete facilita la creación y publicación de representaciones gráficas 2D.
- La URL en PyPI: https://pypi.org/project/matplotlib/
- La URL del proyecto: https://matplotlib.org/
- Durante su instalación, *pip* instalará automáticamente algunas dependencias (si no estuvieran ya instaladas), como la librería de funciones matemáticas *numpy*, con la que está estrechamente vinculada

```
~$ pip3 install matplotlib
```

- Una vez instaladas, podremos importarlas mediante mediante la sentencia import y acceder a sus funcionalidades

PyPI. El repositorio de Python (IV)

```
"""Ejemplo de generación de gráficas."""
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(-2, 2.1, 0.1)   # rango de valores eje-x

# generación de las gráficas
plt.plot(x, 2*x, label="2·x (lineal)")              # f(x) = 2·x
plt.plot(x, x**2, label="x² (cuadrática)")      # f(x) = x²
plt.plot(x, x**3, label="x³ (cúbica)")              # f(x) = x³

# decoraciones
plt.title('Gráficas')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(True)
plt.legend()

plt.show()   # mostramos la gráfica
```

PyPI. El repositorio de Python (y V)