

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

02.01 - Python introducción

Características (I)

Python es un lenguaje de programación potente y fácil de aprender. En líneas generales, podemos considerar este lenguaje como:

- **de alto nivel**, con un fuerte grado de abstracción de las particularidades del hardware, facilitando el empleo del lenguaje natural y, por tanto, su aprendizaje.
- **de propósito general**, diseñado para la escritura de software en cualquier ámbito de aplicación.
- **interpretado**, de forma que sus instrucciones se ejecutan secuencialmente por el intérprete de Python sin necesidad de una compilación previa.

Características (y II)

- Python soporta diferentes **paradigmas de programación** como la programación estructurada o la orientada a objetos
- Es **multiplataforma**. Existen intérpretes de Python para Linux, Windows e Mac OS X
- Es de **tipado dinámico** y dispone de un sistema de **gestión automática de la memoria**.
- Un objetivo de diseño en su desarrollo fue la **extensibilidad**. Dispone de una amplia biblioteca de módulos que nos permiten el desarrollo de aplicaciones destinadas a todo tipo de fines que podemos ampliar con módulos propios o de terceros desarrollados en Python, C ó C++.

Notas históricas (I)

- El desarrollo de Python se inició a finales de los ochenta como un proyecto personal del programador holandés **Guido van Rossum** que, en ese momento, se encontraba trabajando en el Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica) en Holanda.
- El nombre del lenguaje proviene de su afición por la serie Monty Python's Flying Circus de la BBC.
- Tras una publicación interna previa en el CWI, van Rossum publicó en 1991 el código de la versión 0.9.0 en el grupo de noticias **alt.sources** con objeto de abrir el desarrollo de python a la comunidad de programadores.
- El proyecto alcanzaría la versión 1.0 en enero de 1994.

Notas históricas (y II)

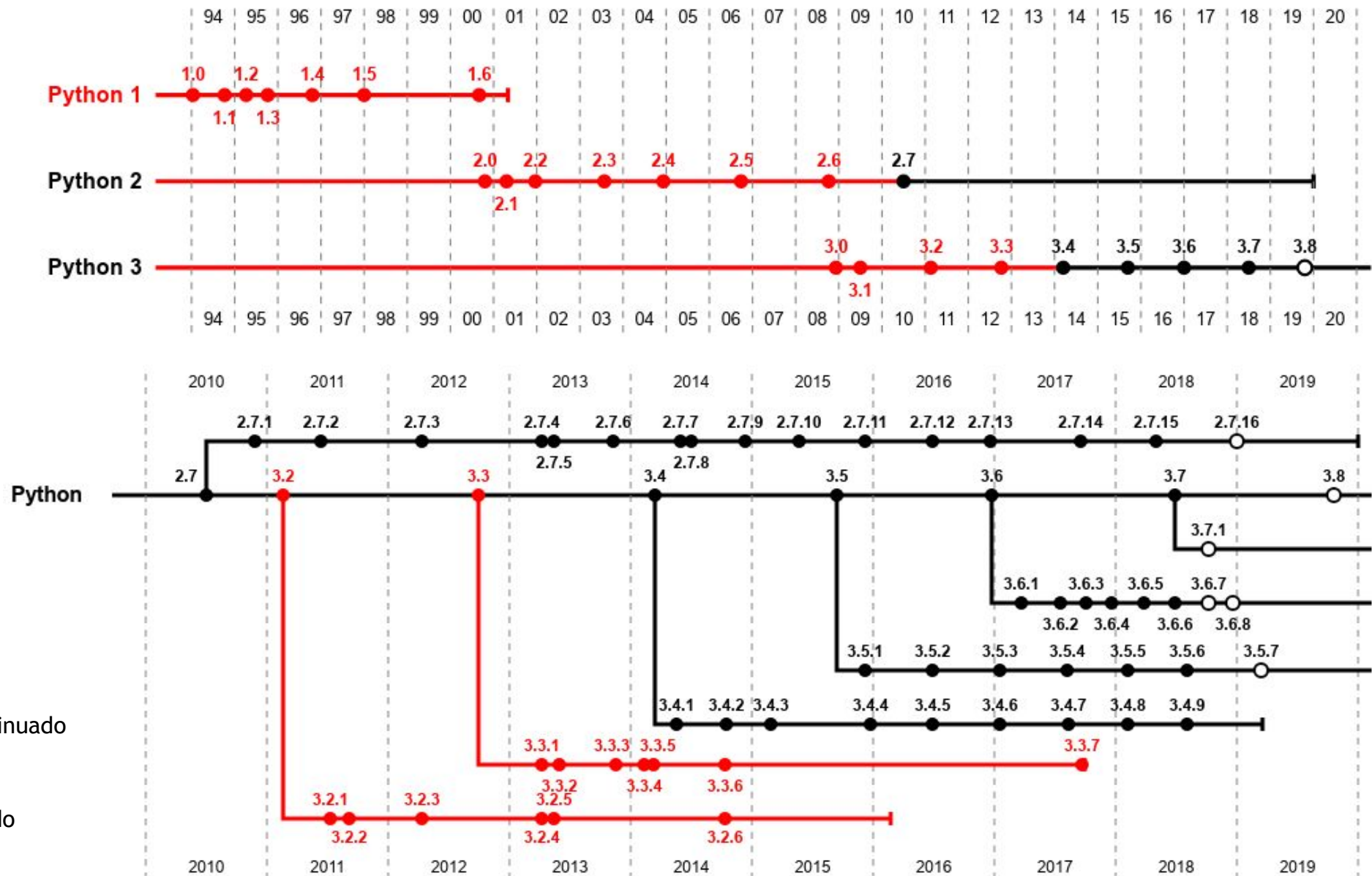
- En la actualidad, el desarrollo y difusión de Python son tareas de la **Python Software Foundation**. Sin embargo, van Rossum continúa ejerciendo un rol central decidiendo la dirección que debe tomar el lenguaje. En la comunidad de Python se le conoce como “Benevolente Dictador Vitalicio”.
- Python es **software libre** licenciado bajo la *Python Software Foundation License*. Esta es una licencia al estilo BSD, más permisiva que la GPL, ya que no requiere que las modificaciones realizadas al código fuente, ni los trabajos derivados, deban ser distribuidos como código abierto.
- Los usuarios de Python se refieren a menudo a la **Filosofía Python** que es bastante análoga a la filosofía de Unix. El código que sigue los principios de Python de legibilidad, simpleza, transparencia y practicidad se dice que es “pythonico”

Versiones (I)

Las versiones de Python se identifican por tres números **X.Y.Z**:

- **X** corresponde a las grandes versiones de Python (1, 2 y 3), siendo incompatibles entre sí. Actualmente se mantienen en desarrollo dos versiones de Python, la **2** y la **3**
- **Y** corresponde a versiones importantes en las que se introducen novedades en el lenguaje pero manteniendo la compatibilidad (salvo excepciones). Suelen publicarse cada año y medio y se mantienen durante cinco años, excepto la versión **2.7** (última que se liberará de la versión 2) que se mantendrá durante diez años hasta 2020
- **Z** se corresponde con revisiones menores que son liberadas durante el periodo de mantenimiento. Corrigen errores y fallos de seguridad.
- Las últimas versiones publicadas de las ramas 2 y 3 son la **2.7.15** (mayo de 2018) y la **3.7.0** (junio de 2018)

Versiones (y II)



Instalación (I)

Linux

- En general, en las distribuciones Linux nos encontraremos con que Python ya se encuentra instalado por defecto en el sistema, bien en su versión 2, en la 3 ó en ambas. Podemos comprobarlo mediante la ejecución de los siguientes comandos:
- Versión 2:

```
zeroth@prog01:~$ python -V  
Python 2.7.9
```

- Versión 3:

```
zeroth@prog01:~$ python3 -V  
Python 3.4.2
```


Instalación (II)

- En caso contrario, los paquetes necesarios para la instalación se encontrarán en el repositorio de la distribución correspondiente y podrán ser instalados con las herramientas disponibles en cada distribución para ello. Por ejemplo, en Debian (como *root*):
- Versión 2:

```
root@prog01:~# apt-get update  
root@prog01:~# apt-get install python idle
```

- Versión 3:

```
root@prog01:~# apt-get update  
root@prog01:~# apt-get install python3 idle3
```

- Notar que, en ambos casos, además de la instalación del intérprete del lenguaje (y las librerías), se ha instalado el entorno de desarrollo IDLE

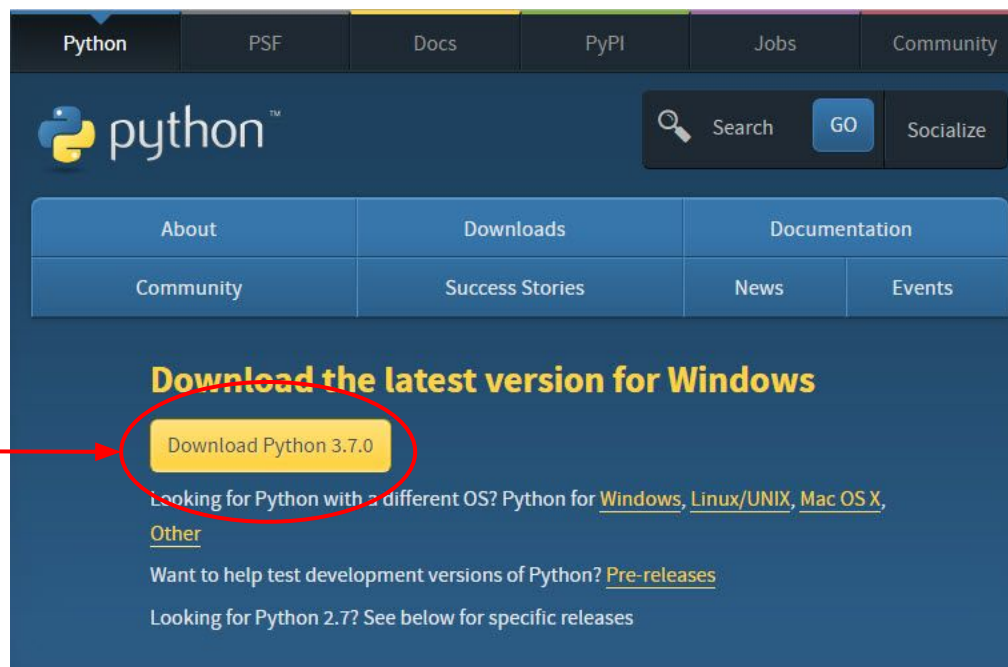
Instalación (III)

Windows

- Las siguientes imágenes muestran el proceso de instalación en Windows 7 de Python 3. El instalador incluye el intérprete y las herramientas IDLE y PIP. El espacio total requerido es de ~100MB

1) Descargar Python de la web oficial: <https://www.python.org/downloads/>

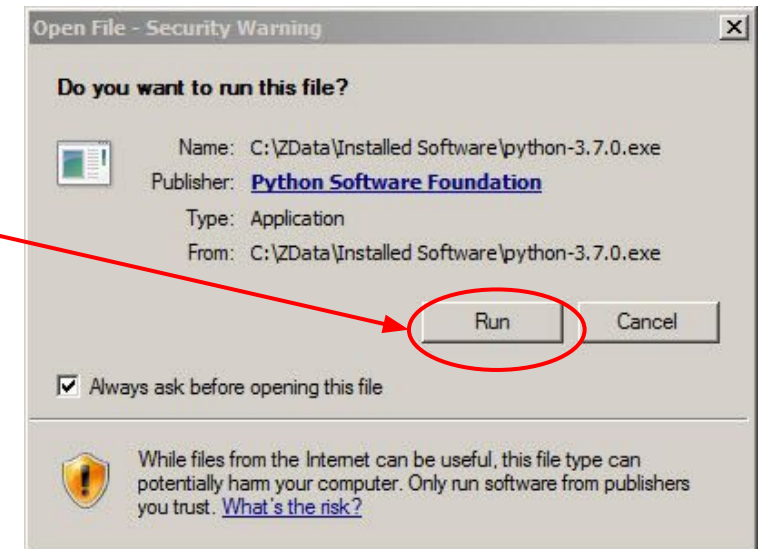
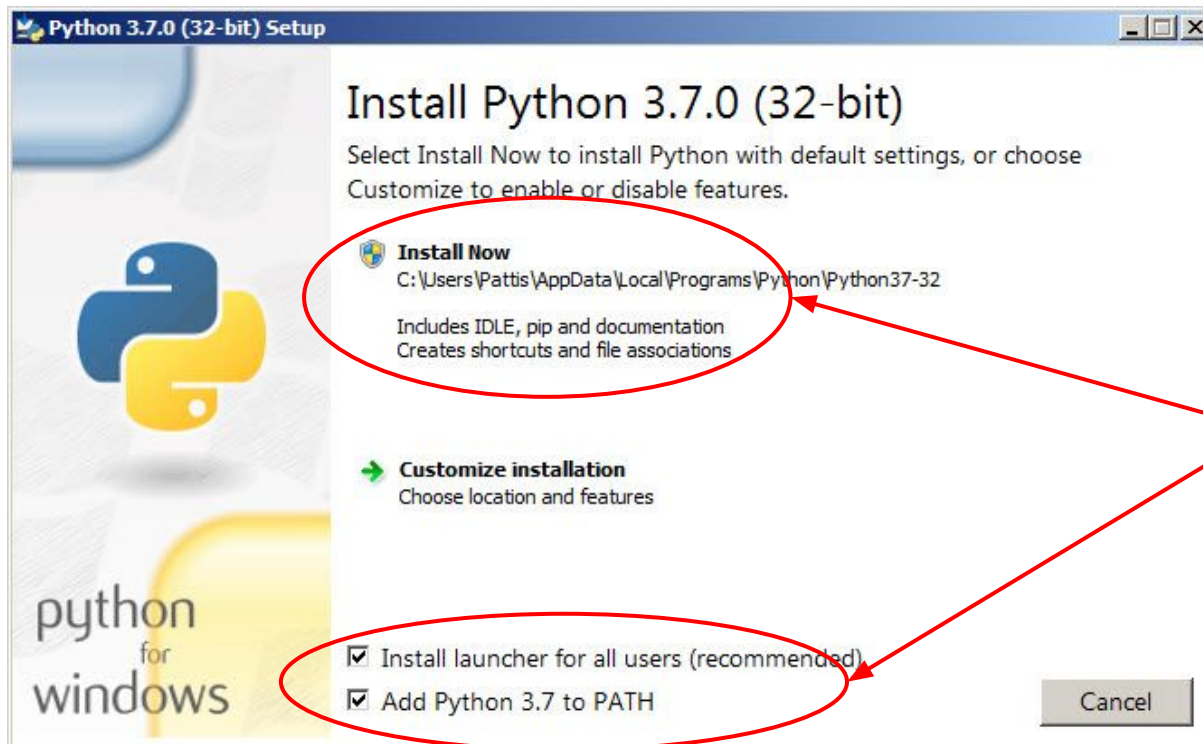
Pulsar para descargar
el instalador
python-3.7.0.exe



Instalación (IV)

2) Ejecutar el fichero **python-3.7-0.exe** del instalador y aceptamos en el aviso de seguridad

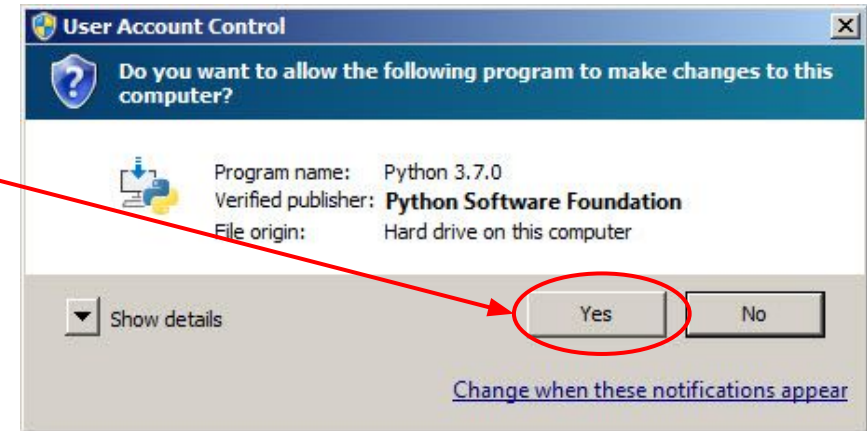
3) Se abrirá la ventana de inicio de la instalación:



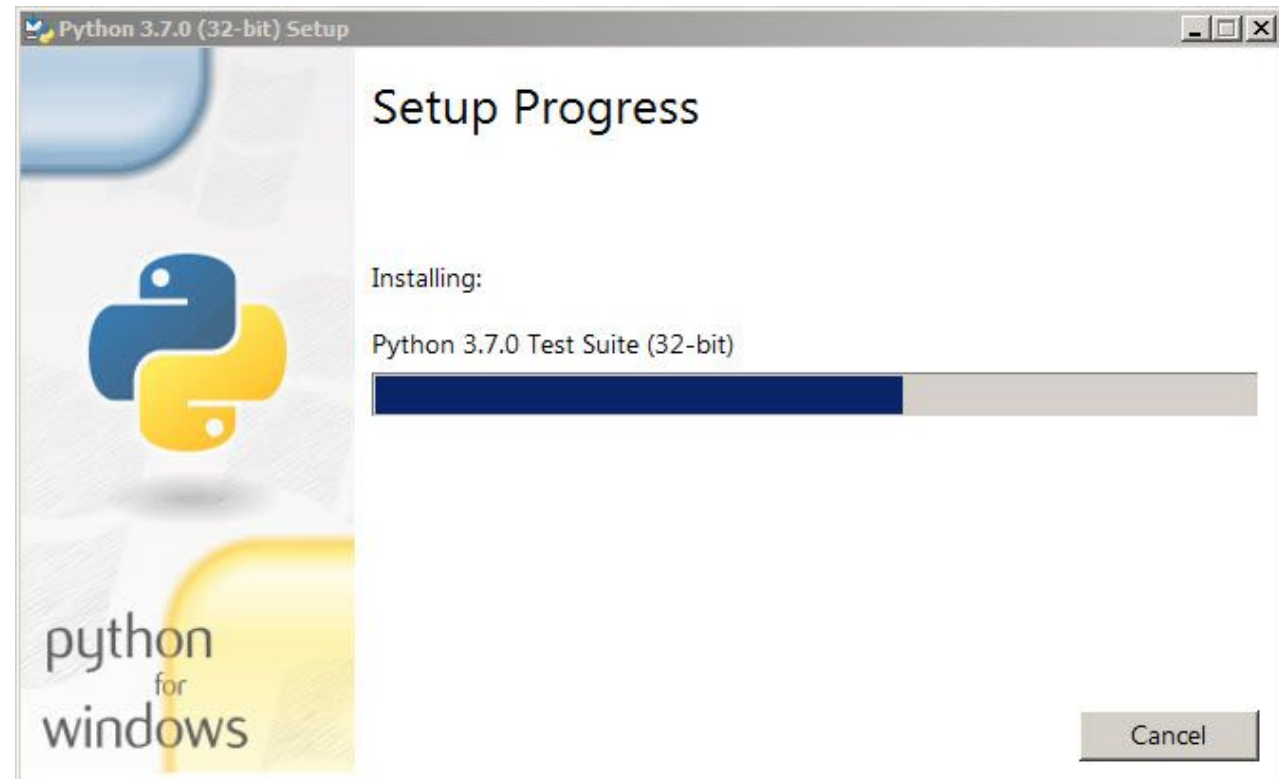
- Verificaremos la carpeta de instalación y que las casillas de “*Instalación para todos los usuarios*” y “*Añadir Python 3.7 al PATH*” estén marcadas.
- Pulsaremos sobre el mensaje “*Install Now*” para continuar

Instalación (V)

4) Se abrirá una ventana solicitando permisos para que la nueva aplicación haga cambios en el sistema. Aceptaremos y ...

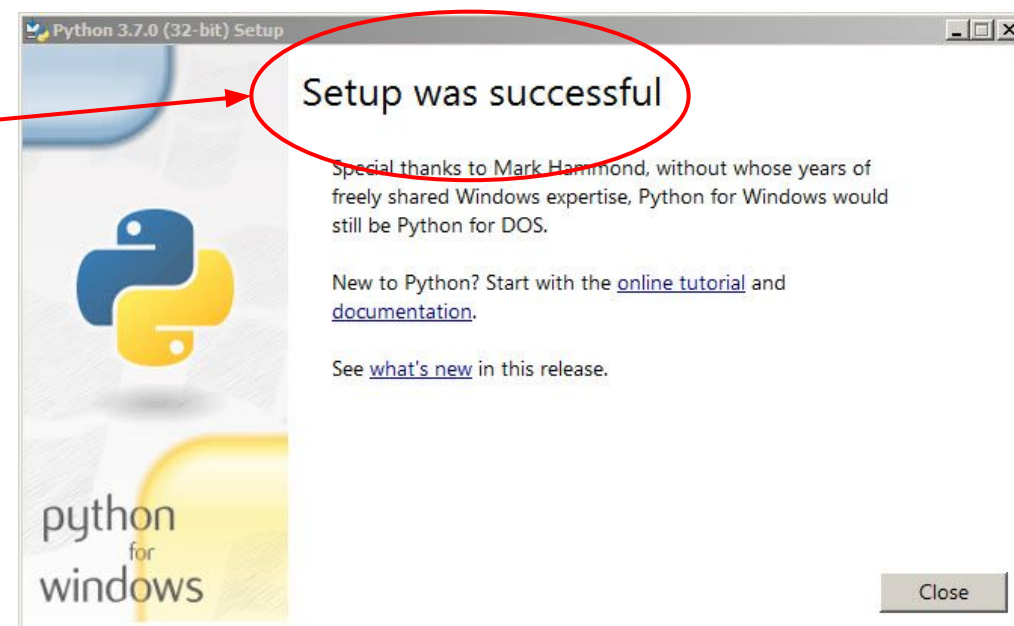


5) Se mostrará la ventana de Progreso de Instalación




Instalación (y V)

6) Por último, se mostrará un mensaje indicado la finalización correcta de la instalación.



Verificación de la instalación

- En el menú de programas se habrá creado una entrada nueva llamada Python. Pulsaremos sobre el icono **Python 3.7 (32-bit)**, que lanzará una consola del intérprete.
- Para verificar que el **PATH** se ha actualizado correctamente, abriremos una consola de Windows y ejecutaremos el comando **python**, que debería lanzar la misma consola

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\Pattis\AppData\Local\Programs\Python\Python37-32\python.exe'. The command prompt shows the text: 'Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32', followed by 'Type "help", "copyright", "credits" or "license()" for more information.' and the prompt '>>>'.

La consola de Python (I)

- Al lanzar el intérprete mediante el comando correspondiente, si no le pasamos como argumento el nombre de un archivo fuente para que lo ejecute, nos mostrará la **consola interactiva** del propio intérprete.

```
zeroth@prog01:~$ python3
Python 3.4.2 (default, Sep 25 2018, 22:02:39)
[GCC 4.9.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- En la primera línea de cabecera se indica la versión de Python que está ejecutando (3.4.2). Tras el **prompt** (>>>), podremos escribir comandos de Python que se ejecutarán al pulsar **Intro**. Si el comando genera algún tipo de resultado, este se mostrará a continuación. Una vez finalizada la ejecución del comando, se mostrará nuevamente el prompt a la espera de una nueva orden.

La consola de Python (II)

- En la consola de Python podremos escribir todo tipo de expresiones algebraicas que serán ejecutadas por el intérprete:

```
>>> 3 + 2
5
>>> 3/2
```

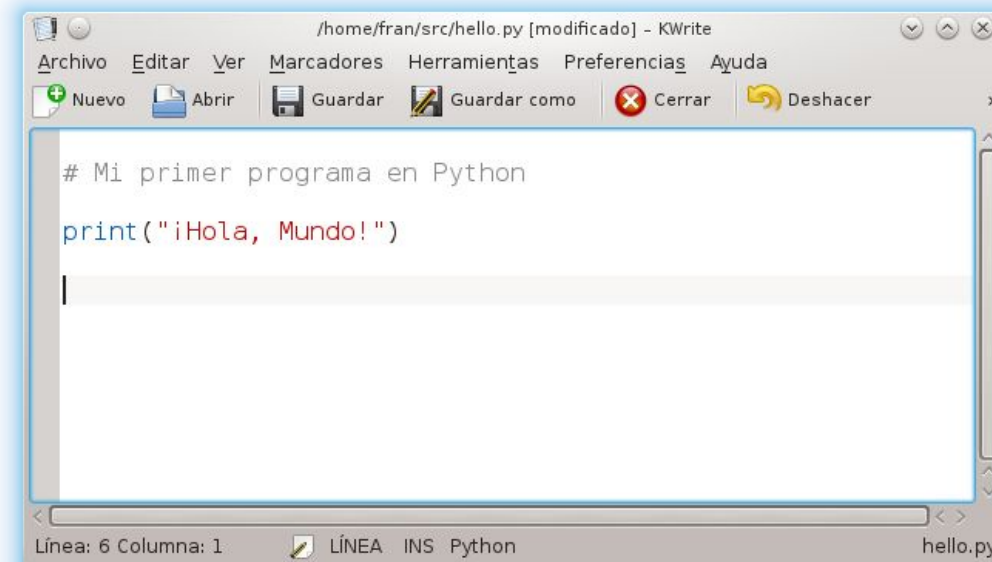
- Podemos ejecutar también instrucciones y funciones de Python:

```
>>> print("¡Hola, Mundo!")
¡Hola, Mundo!
>>>
```

En este ejemplo se hace uso de una función **print** de Python para mostrar el mensaje que se pasa como argumento. El texto del mensaje puede ir encerrado con **entrecorriente** simple o doble.

La consola de Python (III)

- Para crear nuestros programas en Python, sólo necesitaremos un simple editor de textos. A modo de ejemplo, vamos a crear un simple “Hola Mundo”.
- Abrimos un editor y escribimos el código de la imagen. La primera línea comienza con **#** que le indica la intérprete que el texto que va a continuación es un **comentario** para documentar el programa y no debe ser ejecutado. La siguiente línea es una llamada a la función **print** que imprimirá el mensaje pasado como argumento



The screenshot shows a KWrite text editor window titled "/home/fran/src/hello.py [modificado] - KWrite". The menu bar includes Archivo, Editar, Ver, Marcadores, Herramientas, Preferencias, and Ayuda. The toolbar has buttons for Nuevo, Abrir, Guardar, Guardar como, Cerrar, and Deshacer. The text area contains the following code:

```
# Mi primer programa en Python
print("¡Hola, Mundo!")
|
```

The status bar at the bottom indicates "Línea: 6 Columna: 1" and "LÍNEA INS Python". The file name "hello.py" is shown in the bottom right corner.

- El archivo lo guardaremos con la extensión **.py**, para indicar que contiene código fuente Python

La consola de Python (y IV)

- Para ejecutar el código anterior desde la consola, sólo tenemos que lanzar el intérprete indicándole el nombre (y la ruta) del archivo fuente

```
zeroth@prog01:~$ python3 src/hello.py
¡Hola, Mundo!
zeroth@prog01:~$
```

- En Linux, podemos convertir estos archivos en *scripts ejecutables*. Para ello, añadimos la siguiente línea (*hashbang*) al **comienzo** del archivo:

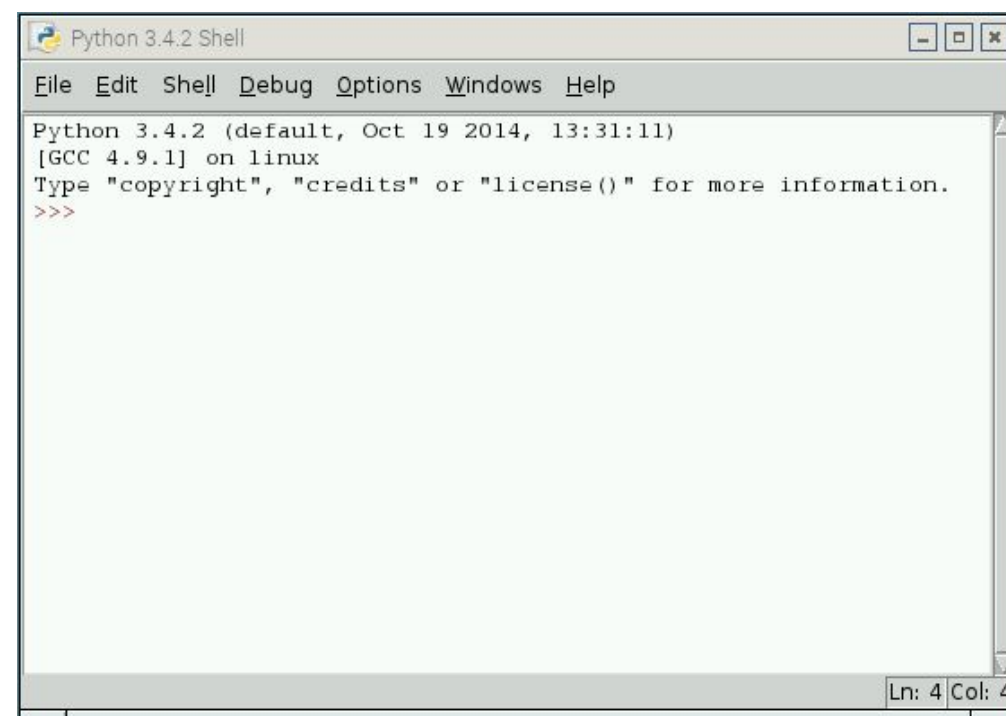
```
#!/usr/bin/env python3
```

- Este *hashbang* le indica al shell que ejecute el intérprete (python3) para procesar el contenido del mismo. Ahora podemos darle permisos de ejecución y lanzarlo como cualquier otro programa

```
zeroth@prog01:~/src$ chmod u+x hello.py
zeroth@prog01:~/src$ ./hello.py
¡Hola, Mundo!
```

IDLE (I)

- Las distribuciones de Python incluyen un IDE simple, denominado IDLE (o IDLE3), para facilitar la escritura, ejecución y depurado de programas en Python
- Podemos lanzar IDLE desde el menú de Python o desde la consola con el comando *idle* (*idle3* para Python 3)
- Al iniciarse nos muestra la típica consola Python, con la diferencia de que incluye resaltado de sintaxis
- Los atajos **Alt+p** y **Alt+n** nos permiten recuperar comandos que introdujimos en la consola con anterioridad.



```
Python 3.4.2 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.2 (default, Oct 19 2014, 13:31:11)
[GCC 4.9.1] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

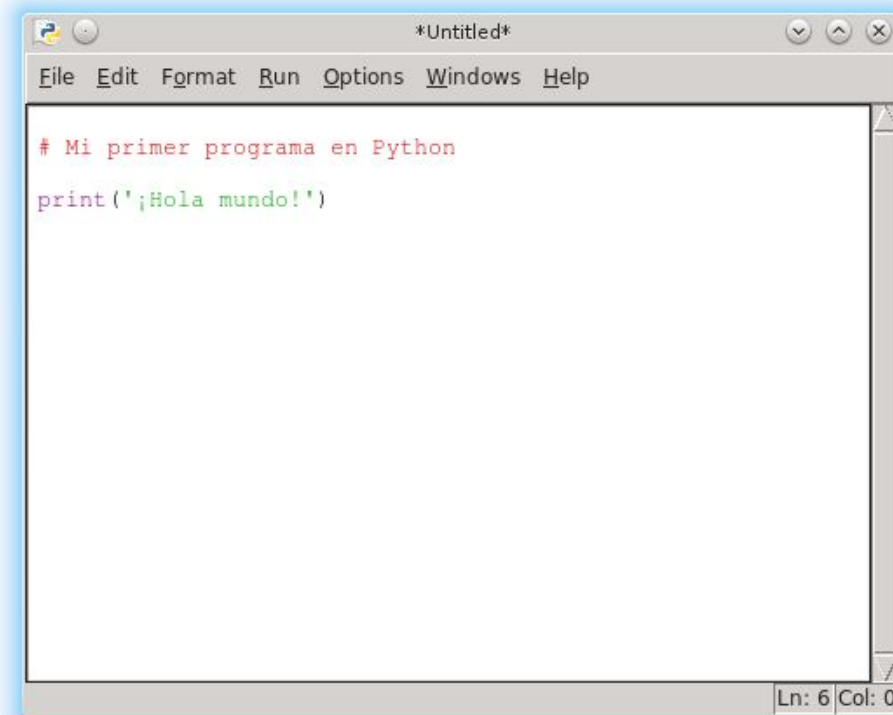
IDLE (II)

- IDLE utiliza **resaltado de sintaxis** para identificar y diferenciar diferentes elementos del lenguaje o errores:
 - Las palabras reservadas de Python (las que forman parte del lenguaje) se muestran en color **naranja**
 - Las cadenas de texto se muestran en **verde**
 - Los resultados de las órdenes se escriben en **azul**
 - Los mensajes de error se muestran en **rojo**
 - Las funciones se muestran en **púrpura**
- El siguiente ejemplo muestra un error al tratar de ejecutar una función (print) sin encerrar sus argumentos (en este caso, el mensaje a mostrar) entre paréntesis:

```
>>> print ";Hola, Mundo!"  
SyntaxError: invalid syntax  
>>>
```

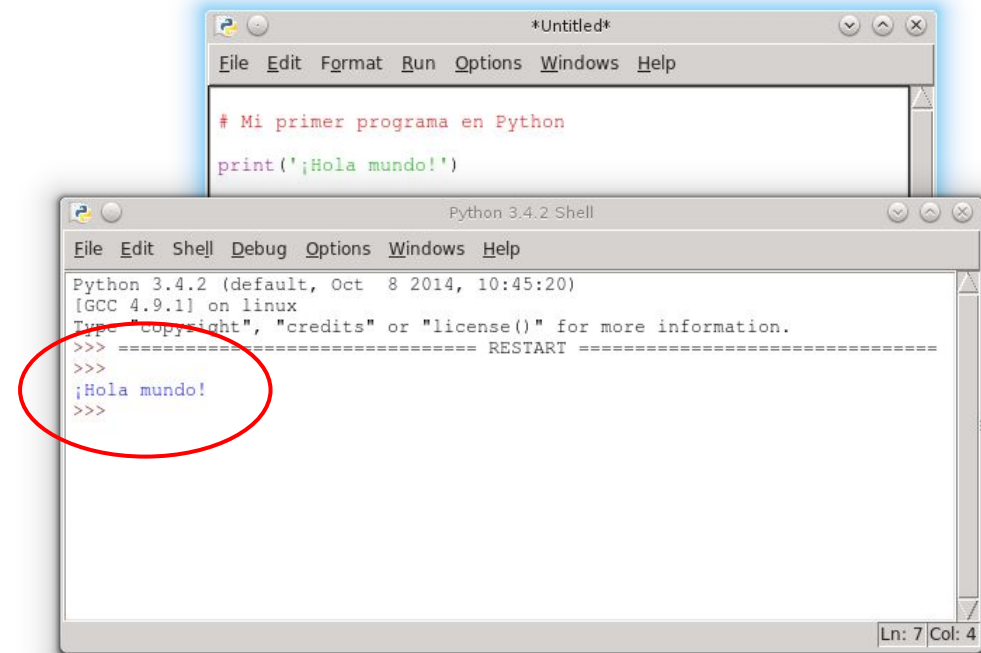
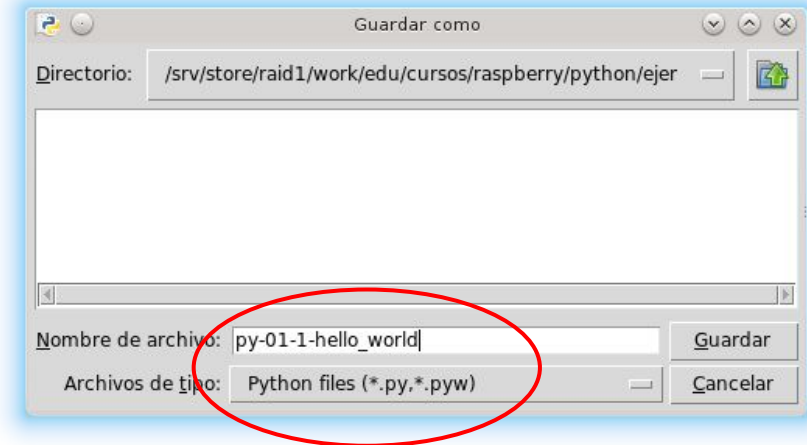
IDLE (III)

- Si bien la consola puede ser útil para realizar pequeñas pruebas, a la hora de programar aplicaciones desearemos que nuestro código se guarde en un archivo que podamos editar posteriormente. Desde el menú *File > New File* (ctrl+n) podremos lanzar una nueva ventana del **editor** de IDLE para crear un nuevo archivo de Python.
- Al escribir nuestro código en la ventana del editor podemos observar que, al igual que pasaba en la consola, se utiliza resaltado de sintaxis



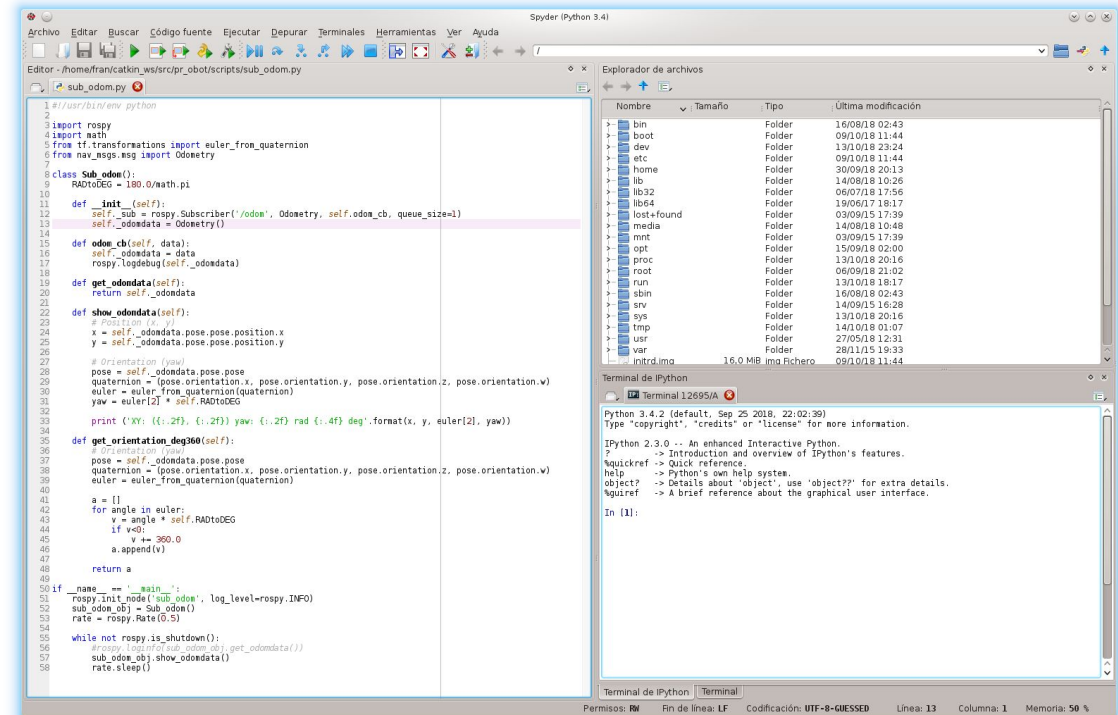
IDLE (y IV)

- Para poder ejecutar nuestro programa, antes deberemos guardarlo. Usaremos la opción *File > Save* (ctrl+S) ó *File > Save As* (ctrl+shift+S) del menú.
- Es importante que el archivo tenga la extensión *.py* para que el editor active el resaltado de sintaxis al abrirlo.
- Para ejecutarlo usaremos la opción *Run > Run Module* (F5)
- En la consola de la ventana principal de IDLE se nos mostrará la salida del programa, en este caso, el mensaje que pasamos a la función print



IDE's avanzados (I)

- A medida que nuestros programas se vuelvan más complejos e incluyen más archivos y librerías, se hace necesario el empleo de un IDE más potente que permita: gestión de proyectos, múltiples pestañas de edición, diseño de interfaces gráficos, depurado avanzado, autocompletado de código,...
- Dos IDE de este tipo son:
 - **Spyder**, IDE libre (MIT license) desarrollado en Python y orientado al campo científico
 - **pyCharm**, IDE escrito en Java, dispone de versiones comercial y libre



IDE's avanzados (y II)

- En el siguiente enlace, se muestra una comparativa de diferentes IDE's:

https://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments

- **Spyder**

Home: <https://www.spyder-ide.org/>

Windows (WinPython): <https://winpython.github.io/>

Linux (Debian/Ubuntu): instalar desde repositorios

- **pyCharm**

Home: <https://www.jetbrains.com/pycharm/>

Descarga (Windows/Linux): <https://www.jetbrains.com/pycharm/download/>

Instalación: <https://www.jetbrains.com/help/pycharm/install-and-set-up-pycharm.html>


```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

02.02 - Python básicos

Tipos de datos

- Con objeto de facilitar el tratamiento de los datos por parte de nuestro programa, **Python**, al igual que el resto de lenguajes de alto nivel, ofrece una serie de **tipos de datos** para categorizarlos.
- Estos tipos de datos definen **cómo** se almacenan internamente dichos datos y **qué** operaciones podemos realizar con ellos.
- En general, se suele distinguir entre tipos de datos **escalares**, que se utilizan para tipos de datos atómicos y unidimensionales, y **no-escalares**, para almacenar datos multidimensionales

<i>Escalares</i>	<i>No-Escalares</i>
<i>int</i> , para enteros <i>float</i> , para reales (aproximación) <i>bool</i> , para valores lógicos (True/False) <i>NoneType</i> , para el valor None	<i>str</i> , para cadenas de caracteres <i>tuple</i> , para tuplas <i>list</i> , para listas <i>dict</i> , para diccionarios <i>complex</i> , para números imaginarios

la función **type()**
permite
obtener el tipo
de un valor o
variable

Datos numéricos (I)

Almacenamiento de números

- En Python se diferencia el almacenamiento de números enteros (**int**), reales en coma flotante (**float**) y complejos (**complex**). La librería estándar contiene tipos de datos numéricos adicionales para fracciones (**fraction**) y números decimales de precisión configurable (**decimal**)
- Los números enteros se almacenan con el tipo **int** y permiten el empleo de valores enteros de cualquier longitud (precisión ilimitada) (Python 3 eliminó la distinción entre los tipos **int** y **long**)
- El tipo **float** utiliza precisión doble (equivalente a **double** de C o Java) y dependerá de la arquitectura (usualmente: IEEE-754 *binary64*)
- El tipo **complex** está formado por dos partes, real (**.real**) e imaginaria (**.imag**), cada una de ellas de tipo **float**

Datos numéricos (II)

- Ejemplos

```
>>> 3 + 2
5
>>> 4/2
2.0
>>> type(3.8)
<class 'float'>
>>> 3 + 2 + 4 + 2.0
11.0
>>> type(1 + 3j)
<class 'complex'>
>>> (1 + 3j).imag
3.0
>>> type((1 + 3j).real)
<class 'float'>
```

En Python3, la división siempre genera un *float*

En Python2 **no**: el tipo del resultado será el de mayor precisión del de los operandos. Si ambos operandos son *int*, el resultado también

- En general, los valores *int* suelen ocupar menos espacio de memoria que los de tipo *float* y las operaciones son más rápidas

Datos numéricos (III)

- **Aproximaciones decimales**
- Debemos tener presente que, debido al espacio limitado de almacenamiento de los formatos IEEE-754 (por ej., *binary64* → mantisa de 52 bits + 1 implícito), las representaciones de determinados números decimales son **aproximaciones**. Por ejemplo, los números decimales 0.5 o 0.25, tienen representación exacta en binario (0.1 y 0.01). Sin embargo, para representar el número 0.1_{10} en binario, necesitaríamos **infinitos** dígitos: 0.000110011001100...
- Esto puede producir algunos resultados inesperados, especialmente al realizar comparaciones:

```
>>> 0.1 + 0.2
0.30000000000000004
>>> (0.1 + 0.2) == 0.3
False
```

Datos numéricos (IV)

- Operadores aritméticos

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
División entera	//	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4

- En general, el tipo de dato del resultado será el de mayor precisión de los operandos (en la división siempre es *float*):

```
>>> type(3 + 2)
<class 'int'>
>>> type(3 - 2.0)
<class 'float'>
```

Datos numéricos (V)

- **Precedencia y asociatividad**

La **asociatividad** y la **precedencia** de los operadores establecen la manera en la que se evalúan las expresiones, especialmente cuando intervienen diversos operandos y operadores

```
>>> 3 + 4 - 2  
5
```

suma y resta tienen la misma precedencia y se asocian por la izquierda, así que el orden en que se resolvería sería: primero $3+4$ y, al resultado, restarle 2

```
>>> 3 + 4 * 2  
11
```

el producto tiene mayor precedencia que la suma y se asocia por la izquierda, así que el orden en que se resolvería sería: primero $4*2$ y, al resultado, sumarle 3

```
>>> (3 + 4) * 2  
14  
>>> (3 + 4) * (4 - (2 + 1))  
7
```

el uso de paréntesis nos permite modificar el orden en que se evalúa la expresión, desde los paréntesis más internos a los más externos

Datos numéricos (VI)

Representaciones de enteros

- Además de en base 10, podemos representar los **literales** enteros en bases binaria, octal y hexadecimal. Para indicar una de estas bases, antepondremos al valor numérico el prefijo correspondiente: **0b** (para binario), **0o** (para octal) y **0x** (para hexadecimal)

```
>>> 0b110101
53
>>> 0b1 + 0b0001
2
>>> 0o10 + 0o10
8
>>> 0xcafe
51966
>>> 6 - 0b11*2 + 0xa
10
```

Datos numéricos (y VII)

Operadores a nivel de bit (*Bitwise Operators*)

Operador	Descripción
&	Y (AND)
	O (OR)
^	O exclusivo (XOR)
~	NOT
>>	Desplazamiento DERECHA
<<	Desplazamiento IZQUIERDA

En decimal		En binario	
Expresión	Resultado	Expresión	Resultado
5 & 12	4	00000101 & 00001100	00000100
5 12	13	00000101 00001100	00001101
5 ^ 12	9	00000101 ^ 00001100	00001001
5 << 1	10	00000101 << 00000001	00001010
5 << 2	20	00000101 << 00000010	00010100
5 << 3	40	00000101 << 00000011	00101000
5 >> 1	2	00000101 >> 00000001	00000010

Valores lógicos (I)

El tipo *booleano*

- Es habitual en los lenguajes de programación la existencia de un tipo de datos lógico o *booleano*. Python proporciona el tipo *bool* que admite dos posibles valores: **True** y **False**. Estos valores se utilizan para representar el resultado de expresiones lógicas (**test de verdad**)

```
>>> 3 > 2
True
>>> type(7<5 or 6>4)
<class 'bool'>
```

- Hay tres operadores lógicos (de menor a mayor precedencia):
 - **or** (“o” lógico): devuelve True si **alguno** de sus dos operandos es True
 - **and** (“y” lógico): devuelve True si sus **dos** operandos son True
 - **not** (“no” lógico, *negación*): devuelve el valor contrario del operando

Valores lógicos (II)

- **Tablas de verdad**

and		
Operandos		Resultado
True	True	True
True	False	False
False	True	False
False	False	False

or		
Operandos		Resultado
True	True	True
True	False	True
False	True	True
False	False	False

not	
Operando	Resultado
True	False
False	True

```
>>> True and False
False
>>> not True
False
>>> True or False and True
True
>>> True or False and not False
True
```

Valores lógicos (III)

- Operadores de comparación

Python proporciona ocho operadores de comparación que devuelven un resultado *booleano* **True** o **False**. Todos tienen la misma precedencia, mayor que la de los operadores lógicos

Operador	Descripción
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor o igual que
==	Igual que
!=	Distinto a
is [not]	Igualdad de objetos
[not] in	Pertenencia a colección

```
>>> 2 < 1
False
>>> 1 < 2
True
>>> 3 < 5 >= 2
True
>>> 1 != 0
True
>>> 2 > 4 or 5 == 5 and 2 <= -2
False
>>> (3 + 4*2 - 11) != 0
False
```

Python la evalúa como:
(3<5) **and** (5>=2)

Operadores

- Tabla de operadores

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
División entera	//	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4
Igual que	==	Binario	—	5
Distinto de	!=	Binario	—	5
Menor que	<	Binario	—	5
Menor o igual que	<=	Binario	—	5
Mayor que	>	Binario	—	5
Mayor o igual que	>=	Binario	—	5
Negación	not	Unario	—	6
Conjunción	and	Binario	Por la izquierda	7
Disyunción	or	Binario	Por la izquierda	8

Variables (I)

Variables y asignaciones

- Las **variables** son elementos o estructuras del lenguaje que posibilitan que nuestros programas almacenen valores para su uso posterior

```
>>> pulgadas_a_metros = 0.0254  
>>> 50 * pulgadas_a_metros  
1,27
```

- En la primera de las líneas anteriores se ha creado una **variable** de nombre *pulgadas_a_metros* y se le ha dado el valor (**asignación**). Al asignar un valor a una variable que no existía (**inicialización**), Python reserva un espacio en la memoria, almacena el valor en él y crea una asociación entre el nombre de la variable y dicha dirección de memoria.



Variables (II)

- Una vez creada la variable, podremos usarla en posteriores sentencias de nuestro programa. En el momento de la ejecución, el nombre de la variable será “sustituido” por el valor que en ese momento tenga la posición de memoria a la que apunta
- Mediante el operador de asignación (=) (no confundir con el operador de comparación ==) podremos cambiar el valor de cualquier variable existente.
- De forma general, la asignación es: *variable = expresión*
 - 1) se evalúa la expresión a la derecha del símbolo igual (=)
 - 2) se guarda el valor resultante en la variable indicada a la izquierda
- La primera operación sobre una variable debe ser la asignación de un valor (**inicialización**). Si se intenta usar una variable no inicializada generará un error de tipo *NameError*

Variables (III)

```
>>> pi = 3.14159265359
```

```
>>> r = 1.23
```

```
>>> perim = 2 * pi * r
```

```
>>> perim  
7.72831792278314
```

```
>>> r = 2
```

```
>>> perim = 2 * pi * r
```

```
>>> perim  
12.56637061436
```

pi



3.14159265359

r



1.23

perim



7.72831792278314

*Creación de pi
(inicialización)*

*Creación de r
(inicialización)*

*Creación de perim
(inicialización)*

r



2

perim



12.56637061436

*asignación de
nuevo valor*

*asignación de
nuevo valor*

```
>>> perim = pi * diametro  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'diametro' is not defined
```

*Error: variable "diametro"
no inicializada*

Variables (IV)

- **Identificadores o nombre de variable**

- El nombre de una variable es su *identificador*. Para que un identificador sea válido, debe estar formado por letras, dígitos numéricos (no puede ser el primer carácter) y/o el carácter de subrayado (_)
- Un identificador no puede coincidir con una de las **palabras reservadas** o clave del lenguaje, es decir palabras que ya tienen un significado predefinido para Python

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

- Python distingue entre mayúsculas y minúsculas. Como guía de estilo se suele emplear nombres representativos en **minúsculas** que, cuando son de dos o más palabras, se unen mediante _

Variables (V)

- **Asignaciones con operador**

- Al igual que muchos otros lenguajes, Python soporta la forma **compacta** de asignación con operador asociado.
- Todos los operadores aritméticos disponen de su versión compacta con asignación. Son de la forma:

operador=

(sin espacio entre el operador y el signo =)

- Estas construcciones permiten reemplazar sentencias como:

contador = contador + 1

por otras más compactas como:

contador += 1

```
>>> a = 5
>>> b = 2
>>> a += 4 * b
>>> a
13
>>> z = 1
>>> z *= 3
>>> z **= 2
>>> z -= 1
>>> z /= 2
>>> z %= 4
>>> z
0
```

Cadenas de Caracteres (I)

El tipo String

- Las denominadas **cadenas de caracteres**, que son secuencias de caracteres (letras, números, espacios, símbolos,...) se emplean para la representación de **información textual**.
- Python dispone del tipo **str** para crear variables que almacenen y operen con cadenas de caracteres
- En Python, las cadenas de caracteres deben ir encerradas entre comilla simple (') o comilla doble (")

```
>>> cadena = "cadena es una variable de tipo String para guardar esta cadena"
>>> cadena
'cadena es una variable de tipo String para guardar esta cadena'
>>> type(cadena)
<class 'str'>
```

Cadenas de Caracteres (II)

- Si queremos que nuestro texto contenga comillas, podemos “*escaparlas*” precediéndolas del carácter (\) o bien, encerrar con comilla doble un texto que contenga comillas simples (y viceversa)

```
>>> texto = "Este texto lleva \"comillas\""
>>> otro_texto = 'Y éste "también"'
>>> texto, otro_texto
('Este texto lleva "comillas"', 'Y éste "también"')
```

- El “*escapado*” se utiliza también para introducir “*códigos especiales*”, como el salto de línea (\n) o el tabulador (\t), para formatear la salida.

```
>>> nuevo_texto = "Una línea\ny otra"
>>> nuevo_texto
'Una línea\ny otra'
>>> print(nuevo_texto)
Una línea
y otra
```

Cadenas de Caracteres (III)

- Una característica de Python en el tratamiento de cadenas es que permite un **triple entrecomillado**, con comilla simple o doble, para mantener en la impresión el mismo formato del texto introducido

```
>>> texto_multilinea = '''
Esto es un
    ejemplo de un
        texto  multilinea
'''
>>> texto_multilinea
'\nEsto es un\n    ejemplo de un\n\t\ttexto\tmultilinea\n'
>>> print(texto_multilinea)

Esto es un
    ejemplo de un
        texto  multilinea

>>>
```

Cadenas de Caracteres (IV)

- Operaciones con cadenas de caracteres
- Python emplea el operador **suma (+)** para **concatenar** cadenas de texto

```
>>> cadena_1 = "Hola, "  
>>> cadena_2 = cadena_1 + "Mundo!"  
>>> print(cadena_2)  
Hola, Mundo!
```

Esto es lo que en programación denominamos **sobrecarga de operadores**. El operador tendrá más de un significado y el compilador (o intérprete) decide en cada momento cuál usar en función del tipo de los operandos.

```
>>> 2 + 2  
4  
>>> "2" + "2"  
'22'  
>>> "2" + 2  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: Can't convert 'int' object to str implicitly
```

Cadenas de Caracteres (V)

- Otro operador aritmético *sobrecargado* que podemos usar en Python con cadenas es el operador **producto** (*).
- Este operador nos permite **repetir** una cadena, a modo de patrón, tantas veces como indique el valor numérico asociado

```
>>> "Hola, " * 5
'Hola, Hola, Hola, Hola, Hola, '
>>> separador = "-" * 60
>>> print(separador)
-----

>>> len(separador)
60
>>> print(separador + "\n" + "\t" * 3 + "CABECERA\n" + separador)
-----
                                CABECERA
-----
```

- **len()** es una *función predefinida*, de las que luego hablaremos, que devuelve el **número de caracteres** de la cadena pasada como argumento

Cadenas de Caracteres (VI)

- **Métodos de cadenas de caracteres**
- Python trata internamente a las cadenas (igual que al resto de tipos primitivos) como **objetos** o instancias de la clase **str**.
Esta clase dispone de numerosos **métodos** que nos permiten realizar todo tipo de operaciones con las cadenas: convertir a mayúsculas o minúsculas, encontrar un carácter, sustituciones, división en *tokens* (*split*),...
- Para invocar cualquiera de estos métodos, la sintaxis será:

objeto_cadena.método(arg1,..., argn)

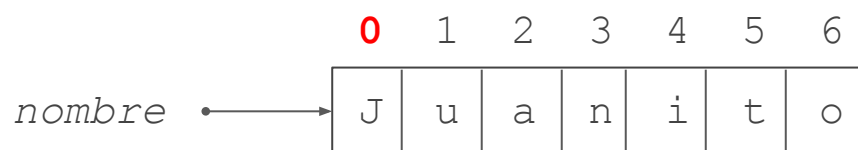
```
>>> cadena_1 = "Hola"
>>> cadena_2 = cadena_1.upper()
>>> print(cadena_2)
HOLA
>>> "Dónde está Wally?".find("Wally")
11
```

Cadenas de Caracteres (VII)

- En la URL <https://docs.python.org/3.7/library/stdtypes.html#string-methods> está la lista completa de métodos de la clase String (<str>). Por ejemplo:
 - **upper()** convierte a mayúsculas, **lower()** a minúsculas, **title()** pasa la primera letra de cada palabra a mayúsculas, **capitalize()** deja el primer carácter en mayúsculas y el resto en minúsculas
 - **find(subcadena[,ini[,fin]])** devuelve la posición de la primera ocurrencia de *subcadena* entre las posiciones *ini* y *fin*
 - **count(subcadena[,ini[,fin]])** cuenta el número de veces que aparece *subcadena* dentro de la cadena entre las posiciones *ini* y *fin*
 - **split([separador])** crea una lista de *tokens* separando mediante el carácter *separador* (“ “ por defecto), **splitlines()** separa por líneas
 - **strip()**, **lstrip()** y **rstrip()** para eliminar espacios en blanco
 - **replace** (*old*, *new*[, *n*]) devuelve una cadena con *n* reemplazos de la subcadena *old* por *new*

Cadenas de Caracteres (y VIII)

- Las cadenas son, en realidad, secuencias de caracteres almacenados en posiciones consecutivas de memoria (*array*). Utilizando el identificador de la cadena y un **índice**, podemos acceder a caracteres individuales o subcadenas (pero no modificar el valor).



Las cadenas en Python
son **inmutables!!**

```
>>> nombre = "Juanito"
>>> nombre[0]
'J'
>>> print(nombre[2])
a
>>> nombre[4:6]
'it'
>>> nombre[4:]
'ito'
>>> "Otra cadena"[:4]
'Otra'
```

El índice del primer carácter es **0**

Extraemos subcadenas con dos índices **[i:j]** i posición inicial, j posición final (no incluida)

Si omitimos i (**[:j]**) o j (**[i:]**), se extrae desde el principio o hasta el final

Funciones predefinidas (I)

- *abs*(*n*), devuelve el valor absoluto del número *n*
- *float*(*exp*), devuelve el número en punto flotante resultado de convertir la expresión *exp*

```
>>> float(5)
5.0
>>> float("5.2")
5.2
```

- *int*(*exp*), devuelve el entero resultado de convertir la expresión *exp*

```
>>> int(5.8)
5
>>> int("-5")
-5
```



“trunca”, no redondea

- *str*(*n*), devuelve la cadena resultante de convertir el número *n*

```
>>> str(3.456)
'3.46'
```

Funciones predefinidas (y II)

- *round*(*n*[,*prec*]), devuelve el entero resultante de redondear *n*. Si se especifica *prec*, devuelve un flotante redondeado a esa precisión (*prec*)

```
>>> round(3.656)
4
>>> round(3.656, 2)
'3.66'
```

- *len*(*cadena*), devuelve el número de caracteres de *cadena*
- *bin*(*n*), *oct*(*n*), *hex*(*n*), devuelven una cadena que representa al número *n* en la base correspondiente
- *ord*(*char*), devuelve el valor numérico del carácter *char*
- *chr*(*n*), devuelve el carácter asociado al código numérico *n*

```
>>> chr(65)
'A'
>>> ord('A')
65
```

Salida formateada (I)

El método *format*

- A la hora de mostrar los resultados de la ejecución de nuestros programas, nos encontraremos con el problema de mostrarlos de la forma más clara y conveniente posible
- La clase `<str>` nos proporciona el método *format()* para producir salidas formateadas
- La cadena a mostrar contendrá una serie de “*campos de reemplazo*” rodeados por llaves `{ }` que serán reemplazados por la lista de argumentos de *format*, de forma que `{0}` es el primer argumento, `{1}` el segundo y así sucesivamente

```
>>> a = 3.5234
>>> 'la suma de {0} + {0} + {1} es {2}'.format(a, 2.0, 2*a + 2.0)
'la suma de 3.5234 + 3.5234 + 2.0 es 9.046800000000001'
```

Salida formateada (II)

- Python nos permite indicar en el *campo de reemplazo* el formato preciso con que queremos que se visualice:

{campo:formato}

formato → *[[relleno] alineamiento] [signo] [#] [0] [ancho] [.precisión] [código de tipo]*

- *relleno*, carácter de relleno para los espacios de alineamiento (por defecto, espacio en blanco)
- *alineamiento*, < izquierda, > derecha (defecto), ^ centrado
- *signo*, + el signo siempre aparece, - sólo para negativos (defecto)
- #, los enteros en binario, octal y hexadecimal se preceden con *prefijo*
- 0, si aparece se usa 0 para sustituir espacios en blanco
- *ancho*, número mínimo de caracteres que ocupará el valor representado
- *.precision*, número de decimales para números en punto flotante
- *código de tipo*, carácter que indica el tipo de representación

Salida formateada (III)

- **Códigos de tipo**

- números enteros

- *b*, binario
 - *c*, carácter Unicode
 - *d*, base diez (defecto)
 - *o*, octal
 - *x*, hexadecimal
 - *n*, como d, pero formato local

- números en coma flotante

- *e*, notación exponente
 - *f*, notación de punto fijo
 - *g*, notación general (defecto)
 - *n*, como d, pero formato local
 - *%*, multiplicado por 100, en formato *f* y seguido de símbolo %

Salida formateada (y IV)

- Ejemplos

```
>>> 'El {0:>10} formateado'.format(123)
'El          123 formateado'
>>> 'El {0:0>10} formateado'.format(123)
'El 00000000123 formateado'
>>> 'El {0:@<10} formateado'.format(123)
'El 123@@@@@@@ formateado'
>>> 'El {0:b} formateado'.format(123)
'El 1111011 formateado'
>>> 'El {0:#b} formateado'.format(123)
'El 0b1111011 formateado'
>>> 'El {0:#x} formateado'.format(123)
'El 0x7b formateado'
>>> 'El {0:_^10.2f} formateado'.format(123.45678)
'El __123.46__ formateado'
>>> 'El {0: _>10.2f} formateado'.format(123.45678)
'El _____123.46 formateado'
>>> 'El {0:.4e} formateado'.format(123.45678)
'El 1.2346e+02 formateado'
```

Entrada de datos (I)

La función *input*

- La función predefinida *input()* permite la captura de la entrada de datos del usuario por el teclado.
- Al llamar a esta función, se detiene la ejecución del programa y todo lo que vaya escribiendo el usuario se irá almacenando en un *buffer*. No se retornará el control al programa hasta que pulse la tecla *Enter*
- La función *input()* retornará todo el contenido del *buffer* de modo que podamos almacenarlo en una variable

```
>>> datos_de_entrada = input()
Hola qué tal
>>> datos_de_entrada
'Hola qué tal'
```


Entrada de datos (y II)

- La función `input()` puede tener como argumento un texto que se mostrará antes de solicitar la entrada del usuario
- Es importante tener en cuenta que el valor devuelto por la función `input()` es una **cadena de texto**, por lo que, dependiendo del caso, deberemos convertirla para poder operar con ella

```
>>> dato1 = input('Introduce un número: ')
Introduce un número: 2
>>> dato2 = input('Introduce otro: ')
Introduce otro: 3
>>> print('La suma de', dato1, 'y', dato2, 'es', dato1 + dato2)
La suma de 2 y 3 es 23
>>> print('Agghhh! No! la suma es', int(dato1) + int(dato2))
Agghhh! No! la suma es 5
```

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=self.sck
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

02.03 - Python

control de flujo

Introducción

Control del Flujo de Ejecución

- Las **sentencias de control del flujo de ejecución** de nuestros programas permiten, bajo determinadas **condiciones**, alterar la ejecución **secuencial** de las instrucciones de nuestros programas.
- Python proporciona diversas instrucciones para controlar flujo de ejecución:
 - ***Sentencias condicionales o de selección:*** **if**
Permiten tomar decisiones en base a los datos y/o resultados y, en función de estos, ejecutar ciertas sentencias y otras no
 - ***Sentencias iterativas o de repetición:*** **for, while**
Permiten tomar decisiones en base a los datos y/o resultados y, en función de estos, ejecutar ciertas sentencias y otras no

Sentencias condicionales (I)

- Supongamos que diseñamos un programa para la resolución de ecuaciones de 1^{er} grado de la forma:

$$a \cdot x + b = 0$$

- Una posible solución podría ser:

```
'''  
ecu_1G_solver.py    ver.1  
Resuelve ecuaciones de 1er grado  
'''  
  
# coeficientes  
a = float(input('Introduce el coeficiente A: '))  
b = float(input('Introduce el coeficiente B: '))  
  
# resultado  
x = -b/a  
print('La solución es x = {0:.2f}'.format(x))
```

Sentencias condicionales (II)

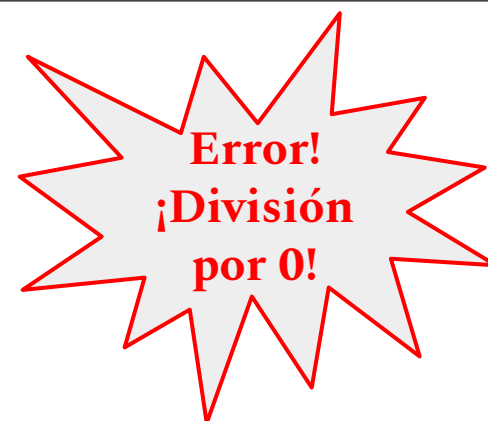
- Vamos a ejecutar nuestro programa:

```
~$ python3 ecu-1G-solver.py  
Introduce el coeficiente A: 2  
Introduce el coeficiente B: 5  
La solución es x = -2.50
```



- Sin embargo, vamos a ejecutarlo de nuevo con estos otros coeficientes:

```
~$ python3 ecu-1G-solver.py  
Introduce el coeficiente A: 0  
Introduce el coeficiente B: 4  
Traceback (most recent call last):  
File "ecu_1G_solver.py", line 10, in <module>  
    x = -b/a  
ZeroDivisionError: float division by zero
```



- Como era de esperar, se ha producido un error al tratar de realizar una división donde el divisor es 0!! Nuestro programa debe controlar esto!!

Sentencias condicionales (III)

La sentencia condicional *if*

- Python nos proporciona la instrucción *if* para evaluar una **condición** y, **sólo en caso de que sea cierta**, ejecutar las acciones que consideremos.

Su formato es:

```
if condición:  
    acción  
    acción  
    ...
```

- condición** podrá ser cualquier *expresión* que devuelva un valor lógico. Si el resultado de evaluar dicha expresión es *verdadero* (**True**), se ejecutarán aquellas *acciones* contenidas dentro del **bloque** de instrucciones delimitado por los dos puntos (:) de *inicio de bloque* y **sangradas** al menos un espacio respecto al inicio de la sentencia *if*. **Todas** las sentencias del bloque deben tener la **misma indentación**.

Sentencias condicionales (IV)

- Vamos a rehacer nuestro programa:

```
'''
ecu_1G_solver.py    ver.2
Resuelve ecuaciones de 1er grado
'''

# coeficientes
a = float(input('Introduce el coeficiente A: '))
b = float(input('Introduce el coeficiente B: '))

# resultado (si a!=0 se ejecutan las instrucciones sangradas a la derecha)
if a != 0:
    x = -b/a
    print('La solución es x = {0:.2f}'.format(x))

# fin programa
print('Fin del programa')
```

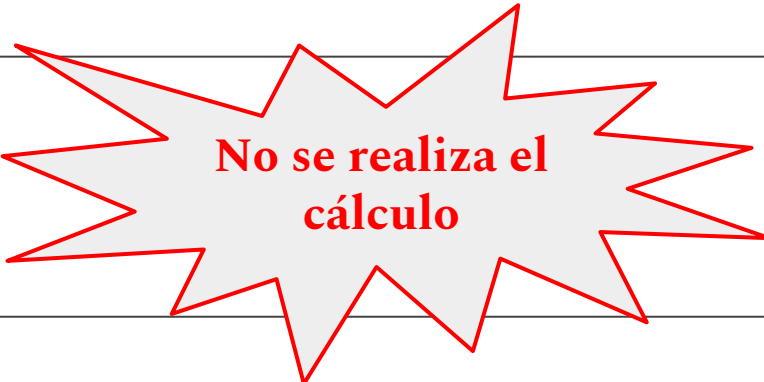
Sentencias condicionales (V)

- Vamos a ejecutar de nuevo nuestro programa:

```
~$ python3 ecu-1G-solver.py  
Introduce el coeficiente A: 2  
Introduce el coeficiente B: 5  
La solución es x = -2.50  
Fin del programa
```

- Sin embargo, vamos a ejecutarlo de nuevo con estos otros coeficientes:

```
~$ python3 ecu-1G-solver.py  
Introduce el coeficiente A: 0  
Introduce el coeficiente B: 4  
Fin del programa
```



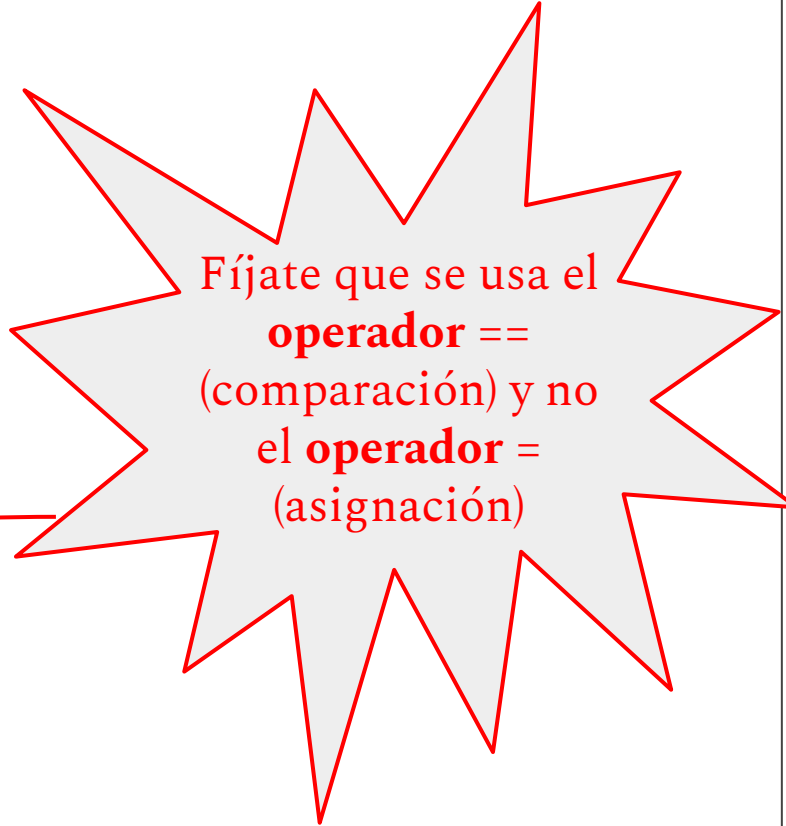
**No se realiza el
cálculo**

- En el segundo caso, al no cumplirse la condición (la expresión $a \neq 0$ devuelve un valor False), se **salta** el bloque de instrucciones incluido en el **if** y se continúa en la siguiente instrucción (`print('Fin del programa')`)

Sentencias condicionales (VI)

- Hemos corregido nuestro error pero deberíamos informar de dicha situación al usuario en lugar de finalizar abruptamente

```
'''  
ecu_1G_solver.py    ver.3  
Resuelve ecuaciones de 1er grado  
'''  
  
a = float(input('Introduce el coeficiente A: '))  
b = float(input('Introduce el coeficiente B: '))  
  
if a != 0:  
    x = -b/a  
    print('La solución es x = {0:.2f}'.format(x))  
  
# si a es 0 se informa al usuario  
if a == 0:  
    print('La ecuación no tiene solución')  
  
print('Fin del programa')
```



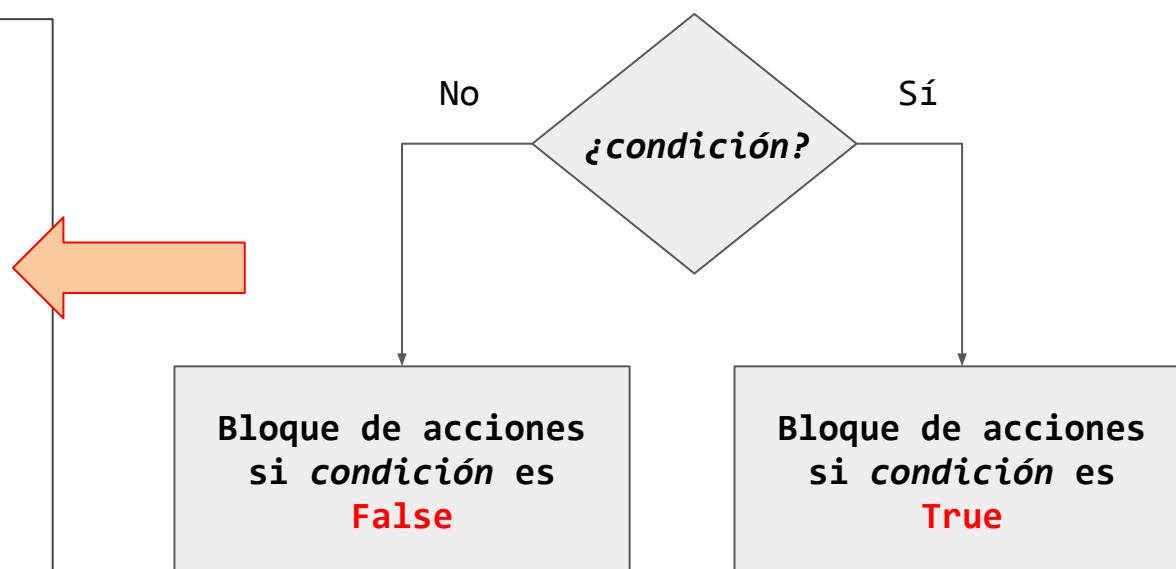
Fíjate que se usa el
operador ==
(comparación) y no
el **operador =**
(asignación)

Sentencias condicionales (VII)

La construcción *if - else*

- Para situaciones, como en el caso anterior, donde nos encontramos con que debemos realizar unas determinadas acciones en caso de que se verifique una condición y, otras acciones diferentes en el caso contrario, Python nos proporciona la construcción *if - else* que responde al siguiente formato:

```
if condición:  
    bloque de acciones si  
    condición es True  
else:  
    bloque de acciones si  
    condición es False
```



Sentencias condicionales (VII)

- Reescribamos nuestro programa de ecuaciones de 1^{er} grado...

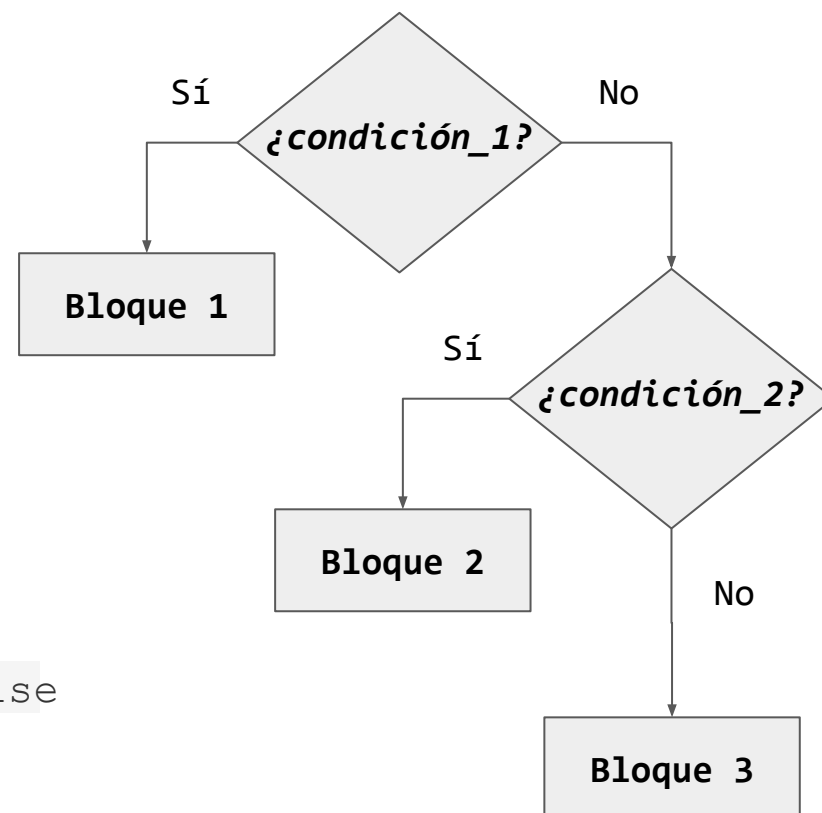
```
'''  
ecu_1G_solver.py    ver.4  
Resuelve ecuaciones de 1er grado  
'''  
  
a = float(input('Introduce el coeficiente A: '))  
b = float(input('Introduce el coeficiente B: '))  
  
if a != 0:  
    x = -b/a  
    print('La solución es x = {0:.2f}'.format(x))  
else:  
    # si a es 0 se informa al usuario  
    print('La ecuación no tiene solución')  
  
print('Fin del programa')
```

Sentencias condicionales (VII)

Condicionales anidados

- Pueden darse casos en los que necesitamos “insertar” condicionales dentro de otros (*anidamiento*), creando estructuras como la siguiente:

```
if condición_1 :  
    # Se ejecuta si condición_1 es True  
    bloque 1  
else :  
    # Se ejecuta si condición_1 es False  
    if condición_2 :  
        # Se ejecuta si condición_2 es True  
        bloque 2  
    else :  
        # Se ejecuta si ambas condiciones son False  
        bloque 3
```



Sentencias condicionales (VII)

- Volvamos al ejemplo. Queremos que, si a es 0, distinguir cuando b es 0 (infinitas soluciones) y b es distinto de 0 (no tiene solución)

```
'''
ecu_1G_solver.py    ver.5
Resuelve ecuaciones de 1er grado
'''

a = float(input('Introduce el coeficiente A: '))
b = float(input('Introduce el coeficiente B: '))

if a != 0:
    x = -b/a
    print('La solución es x = {0:.2f}'.format(x))
else:
    if b == 0:
        print('Indeterminado: la ecuación tiene infinitas soluciones')
    else:
        print('La ecuación no tiene solución')
print('Fin del programa')
```

Sentencias condicionales (VIII)

Condicionales anidados: *if - elif - else*

- Por último, para estos anidamientos de condicionales, Python nos ofrece una construcción del lenguaje más “*elegante*”, la sentencia *elif*, que surge de la **contracción** de *else* con el *if* posterior

```
if condición_1 :  
    # Se ejecuta si condición_1 es True  
    bloque 1  
elif condición_2 :  
    # Se ejecuta si las condiciones anteriores son False y condición_2 es True  
    bloque 2  
elif condición_3 :  
    # Se ejecuta si las condiciones anteriores son False y condición_3 es True  
    bloque 3  
else :  
    # Se ejecuta si todas las condiciones anteriores son False  
    bloque 4
```

Sentencias condicionales (IX)

- Así, la versión definitiva de nuestro ejemplo sería:

```
'''  
ecu_1G_solver.py    ver.6  
Resuelve ecuaciones de 1er grado  
'''  
  
a = float(input('Introduce el coeficiente A: '))  
b = float(input('Introduce el coeficiente B: '))  
  
if a != 0:  
    x = -b/a  
    print('La solución es x = {0:.2f}'.format(x))  
elif b == 0:  
    print('Indeterminado: la ecuación tiene infinitas soluciones')  
else:  
    print('La ecuación no tiene solución')  
  
print('Fin del programa')
```

Indentación en Python

- Una nota final sobre la *indentación* o *sangrado* que usa Python para definir *bloques* de instrucciones. Si bien puede hacerse un tanto extraño al principio, especialmente a programadores que vienen de otros lenguajes como C o Java, donde se usa { } para definir dichos bloques, tiene como claro objetivo obligar y facilitar la *legibilidad* del código.
- Fíjate en el código del siguiente programa. Es nuestro ejemplo pero escrito en lenguaje C. Está un poco "ofuscado" a propósito (de hecho se podría haber escrito todo el programa en un par de líneas)

```
#include <stdio.h>
void main(){ float a,b,x;
printf("Coef A: "); scanf("%f", &a); printf("Coef B: "); scanf("%f", &b);
if(a!=0) {printf("La solución es x=%.2f", -b/a);} else {if(b==0)
{printf("Indeterminado");} else {printf("Sin solución");}}
printf("\nFin programa");}
```

- Guía de estilo de Guido van Rossum (<https://www.python.org/dev/peps/pep-0008/>)

Sentencias iterativas (I)

La sentencia *while*

- Los bucles *while* son los tipos más simples de bucle. En ellos se establece una condición y, mientras esa condición se cumpla (devuelve un valor **True**), el bloque de instrucciones asociado al bucle *while* continuará ejecutándose

- Su formato es:

```
while condición:  
    acción  
    acción  
    ...  
    acción
```

- *condición* será cualquier expresión evaluable que devuelva un valor booleano **True** o **False**

Sentencias iterativas (II)

- El siguiente ejemplo usa un bucle *while* para crear un contador. En cada **iteración** del bucle, se irán ejecutando cada una las instrucciones del bloque *while*. Tras ejecutar la última, se volverá a **comprobar** la condición. **Si se cumple**, se realizará una nueva iteración. En caso contrario, se continuará con la instrucción siguiente al bloque *while*

```
from time import sleep
```

```
print('Se va a iniciar la cuenta atrás...')
```

```
sleep(3)
```

```
temp = 10
```

```
while temp>0:
```

```
    print(temp)
```

```
    sleep(1)
```

```
    temp -= 1
```

```
print('\aDESPEGUE!!!\n')
```

Al **principio** de cada **iteración**,
comprobamos el valor de **temp**

Si se cumplió la condición (**temp>0**),
ejecutamos las instrucciones del bucle

El proceso sigue
repitiéndose
hasta que deje
de cumplirse la
condición

Esta instrucción **sólo** se ejecutará una vez que se salga del bucle

Sentencias iterativas (III)

El bucle *for-in*

- Python dispone de otro tipo de bucle, el bucle *for-in*, que se puede leer como “*para cada elemento de la serie, hacer...*”
- Suele emplearse cuando conocemos de antemano el número exacto de iteraciones que queremos realizar o bien, cuando queremos recorrer secuencialmente una serie de elementos o valores

- Su formato es:

```
for variable in serie_de_valores:  
    acción  
    acción  
    ...  
    acción
```

- En cada *iteración* del bucle, *variable* irá tomando uno de los valores de *serie_de_valores*. Habrá tantas iteraciones como valores en la serie

Sentencias iterativas (IV)

- El siguiente ejemplo usa un bucle *for* para imprimir los nombres contenidos en una lista:

```
for nombre in ['Pedro', 'Juan', 'María']:  
    print(';Hola ' + nombre + '!')  
print('Bienvenidos al curso')
```

- El siguiente ejemplo usa la función *range*, que nos permite generar una secuencia de valores entre un valor inicial y otro final, para calcular el factorial de un número introducido por el usuario:

```
num = int(input('Introduce un número: '))  
  
fact = 1  
for i in range(2, num):  
    fact *= i  
fact *= num  
  
print('{0}! = {1}'.format(num, fact))
```

range(2, num) genera una lista de valores desde 2 hasta num-1
Se realizará una iteración del bucle por cada valor de esa lista

Sentencias iterativas (V)

La función *range*

- La función *range* es lo que en Python se denomina un *generador*. Es decir, produce una secuencia de valores entre unos límites.
- Su formato es: `range([inicio,]final[,paso])`
 - inicio, (*por defecto*, 0) valor inicial del generador
 - final, establece el valor final del generador (**no se incluye**)
 - paso, (*por defecto*, 1) paso o salto entre los valores generados

```
>>> list(range(2, 10))
[2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(4))
(0, 1, 2, 3)
>>> list(range(-3, 3))
[-3, -2, -1, 0, 1, 2]
>>> list(range(0, 10, 2))
[0, 2, 4, 6, 8]
```

Sentencias iterativas (VI)

- En general, la sentencia *for* nos va a permitir recorrer cualquier tipo de objeto **secuenciable** (*iterable*), es decir, objetos que nos permiten acceder de forma secuencial a cada uno de sus elementos. Objetos de este tipo son las cadenas de caracteres, las listas, las tuplas, ...

```
>>> conjunto_1 = (1, 2, 3, 4, 5) # una tupla
>>> conjunto_2 = ['Tengo', '23', 'años'] # una lista
>>> acum = 0
>>> for val in conjunto_1:
    acum += val
>>> print('La suma de los valores de conjunto_1 es', acum)
La suma de los valores de conjunto_1 es 15
>>> for val in conjunto_2:
...     print(val, end=' ')
Teño 23 años
>>> for val in conjunto_2[2]:
    print('[' + val + ']', end=" ")
[a] [ñ] [o] [s]
```

Sentencias iterativas (VII)

break, continue y pass

- En ocasiones se darán casos en que precisemos alterar la ejecución normal dentro del bucle o abandonarlo. Las sentencias *break* y *continue* nos permiten modificar el flujo normal de ejecución de los bucles creados con *while* y *for*

break

- La sentencia *break* fuerza la salida del bucle en el que nos encontremos:

```
>>> while True:
...     val = input("Pulsa [S] para Salir ")
...     if val == 'S':
...         break
```

```
Pulsa [S] para Salir: a
```

```
Pulsa [S] para Salir: S
```

```
>>>
```

Sentencias iterativas (VIII)

- El siguiente ejemplo, que calcula los números primos entre 100 y 200, muestra el uso de *break* para evitar cálculos innecesarios. Para cada uno de estos números, el programa buscará la existencia de algún divisor. En caso de encontrar alguno, ya sabemos que el número no es primo y podemos pasar al siguiente.

```
print('Los números primos entre 100 y 200 son:')

for numero in range(100, 201):
    primo = True
    for divisor in range(2, numero):
        if numero%divisor == 0:
            # no es primo -> saltar al siguiente
            primo = False
            break

    # si numero es primo, lo imprimimos
    if primo:
        print(numero, end=' ')
```

FÍJATE

Podemos anidar bucles, condicionales,... *break* y *continue* sólo afectarán al bucle más interno donde se encuentren

IMPORTANTE

Vigila el *sangrado* para que Python identifique correctamente cada bloque


Sentencias iterativas (IX)

continue

- La sentencia *continue* dentro de un bucle, provoca que se salte al inicio de la siguiente iteración sin ejecutar el resto de instrucciones del bucle. Es decir, no salimos del bucle (como con *break*), si no que saltamos al inicio del mismo (y si se cumple la condición se ejecutará la siguiente iteración)

```
# Imprime número impares entre 1 y 10
print('Números impares menores que 10: ', end='')
for num in range(1, 11):
    if num%2 == 0:
        continue
    print(num, end=' ')
```

Si **num** es **par**, salta al principio del bucle (*continue*) y la sentencia *print* no se ejecuta



output:

```
Números impares menores que 10: 1 3 5 7 9
```

Sentencias iterativas (X)

pass

- La sentencia *pass* representa una operación **nula**, es decir, no ejecuta **nada!!**
- Si bien su existencia puede parecer un tanto absurda, se usa cuando una sentencia es necesaria sintácticamente, pero no necesitamos ejecutar ningún código.
- Es habitual encontrarla en las fases iniciales de nuestro programa cuando tenemos la estructura del código pero aún está sin implementar determinada funcionalidad

```
# recorro el bucle
for val in lista_de_valores:
    if val==valor_erroneo:
        # sé que tengo que hacer algo aquí pero aún no sé qué hacer
        pass
```

Gestión de errores (I)

Las excepciones

- Hemos visto como, bajo determinadas condiciones, pueden aparecer errores en **tiempo de ejecución**, es decir, se pueden generar **excepciones**: divisiones por cero, operaciones con tipos incompatibles, empleo de variables no definidas,...

```
>>> print("'sumar' un String y un número genera un error" + 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

- Estas excepciones provocarán la finalización abrupta e inesperada de nuestro programa, algo que en general no es deseable. Si bien la sentencia **if** nos permite controlar estas situaciones, Python nos proporciona un mecanismo más elegante para gestionar los errores. Se trata de la estructura de control **try-except**

Gestión de errores (II)

La estructura *try-except[-else[-finally]]*

- Veamos el siguiente programa que nos permite determinar si un número es primo o no:

```
num = int(input("Introduce un número: "))  
if num%2 == 0:  
    print(num, "es par")  
else:  
    print(num, "es impar")
```

- Básicamente no hace más que convertir la entrada (de tipo String) y evaluar el resto de la división entera entre 2. Pero, y si no le damos un número! Se genera una excepción *ValueError*

```
Introduce un número: blabla  
Traceback (most recent call last):  
...  
ValueError: invalid literal for int() with base 10: 'blabla'
```



**Error! en la
conversión**

Gestión de errores (III)

- Una forma de solucionar el problema anterior sería utilizar el método *.isdecimal()* de String para evaluar la entrada del usuario...

```
num = input("Introduce un número: ")
if num.isdecimal():
    if int(num)%2 == 0:
        print(num, "es par")
    else:
        print(num, "es impar")
else:
    print("valor no válido")
```

- Antes de realizar la conversión, comprobamos que el valor introducido por el usuario sea un número (en realidad un número natural!). Sólo en ese caso, procederemos a la conversión de String a entero

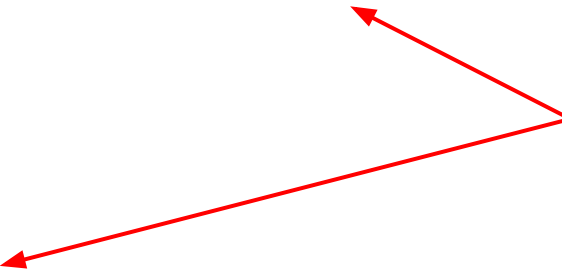
```
Introduce un número: blabla
valor no válido
```

Gestión de errores (IV)

- Veamos cómo lo solucionaríamos empleando *try-exception*
Dentro del bloque *try*, “insertaremos” la sección de código susceptible de generar un posible error y, añadiremos una sección *except* para tratar el error en caso de que se produzca (en este caso, *ValueError*)

```
try:
    num = int(input("Introduce un número: "))
    if num%2 == 0:
        print(num, "es par")
    else:
        print(num, "es impar")
except ValueError:
    print("valor no válido")
```

Si se produce un **error** en la conversión, la ejecución del programa salta al bloque *except* encargado de la gestión de dicha **excepción**



```
Introduce un número: blabla
valor no válido
```

Gestión de errores (V)

- Aunque pudiera parecer que la sentencia *if* nos permite controlar este tipo de situaciones, la realidad es que, en la mayoría de los casos, su uso implica una mayor complejidad o soluciones poco robustas. En el caso anterior, la solución empleando *if*, no nos serviría para números negativos, pues los métodos *isdigit()*, *isnumeric()* o *isdecimal()* de String no interpretan el signo '-' como numérico (lo mismo ocurre con el '.' de los números reales). El programa no generaría un error, pero nos informaría de que el valor "-5" no es válido.

```
Introduce un número: -5  
valor no válido
```

- Evidentemente el uso de *try-except* no es obligatorio. Podríamos resolver el problema empleando *expresiones regulares* o analizando cada carácter del String, pero sería más laborioso y menos *pythónico!!*

Gestión de errores (VI)

- El formato general de *try-except* es el siguiente:

```
try:
    acciones que pueden generar un error
except tipo_de_error_1:
    acciones a ejecutar si se produce tipo_de_error_1
except tipo_de_error_2:
    acciones a ejecutar si se produce tipo_de_error_2
. . .
except :
    acciones a ejecutar si se produce un error no contemplado
else:
    acciones a ejecutar si no se produce ningún error
finally:
    acciones a ejecutar siempre (con o sin error)
```


Gestión de errores (VII)

- Se ejecuta el bloque de instrucciones de la cláusula *try*
- Si no se produce ningún error, todas las cláusulas *except* se saltan y la ejecución continúa según el siguiente orden: cláusula *else* (si existe), cláusula *finally* (si existe) y se sale de la estructura de gestión de errores *try-except* para continuar en la siguiente instrucción
- Si se produce un error, el orden de ejecución será: cláusula *except* del *tipo_de_error* correspondiente, cláusula *finally* (si existe) y se sale de la estructura de gestión de errores *try-except* para continuar en la siguiente instrucción
- Tanto *else*, como *finally*, como el *tipo_de_error*, son opcionales
- En caso de tener una cláusula *except* sin el *tipo_de_error*, se ejecutaría si se produjera un error y no existiera un *except* específico para él
- De existir, las instrucciones de *finally* siempre se ejecutan antes de abandonar la estructura *try-except*, hubiera o no errores

02.03 - Python. Control de Flujo

Gestión de errores (VIII)

- Algunos ejemplos:

```
>>> while True:
...     try:
...         num = int(input("Introduce un número: "))
...         break
...     except ValueError:
...         print("Ooops! Prueba otra vez...")
... print("El número es", num)
```

No se abandona el bucle hasta que se introduzca un número válido

```
>>> try:
...     num1 = float(input("Dividendo: "))
...     num2 = float(input("Divisor: "))
...     result = num1/num2
... except ValueError as err:
...     print(err)
... except ZeroDivisionError:
...     print("Error: división entre 0")
... else:
...     print("{0}/{1} = {2:.3f}".format(num1, num2, result))
```

Dependiendo del error se ejecutará la cláusula *except* correspondiente

Si no hay errores, se ejecuta la cláusula *else*

02.03 - Python. Control de Flujo

Gestión de errores (y IX)

- El siguiente programa suma los números leídos desde un fichero:

```
suma = 0

try:
    fichero = open("numeros.txt") # abrimos el fichero con los valores
    linea = fichero.readline()    # lee una línea del archivo
    while linea != "":
        num = float(linea)        # convertimos el valor leído a número
        suma += num
        linea = fichero.readline() # lee una nueva línea del archivo
except ValueError:
    print("'{}' no se puede convertir a float".format(linea.strip()))
except Exception as err:         ← capturamos cualquier error que no sea ValueError
    print("Error inesperado:", err)
else:                             ← else se ejecuta si no hubo ningún error
    print("El total es {0:.2f}€".format(suma))
finally:                          ← finally siempre se ejecuta,
    fichero.close()               # cerramos el fichero
                                   hubiera o no errores

print("Fin del programa")
```

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

02.04 - Python funciones

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.

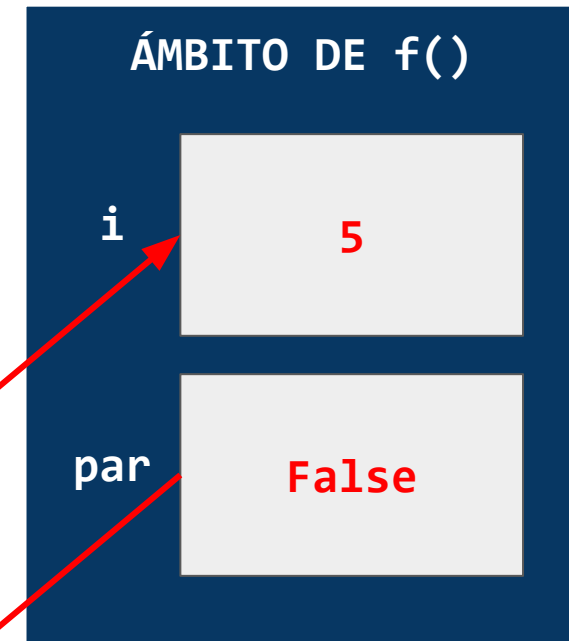
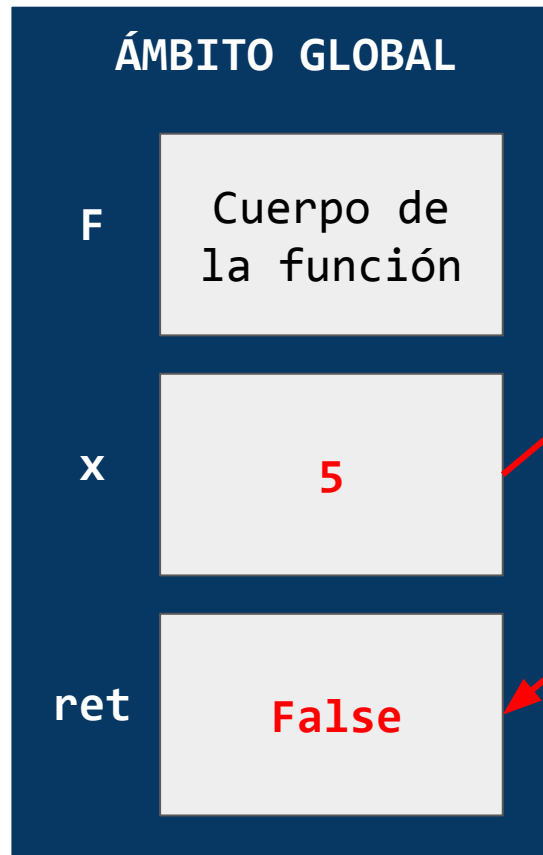
    Args:
        i (int): número entero.
    Returns:
        (bool): True si es par.
    """
    if i%2 == 0:
        par = True
    else:
        par = False
    return par
```

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 siguientes funciones son equivalentes:

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

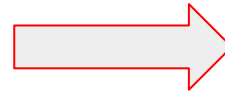
```
def es_par(i):  
    return i%2 == 0
```

- 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
```

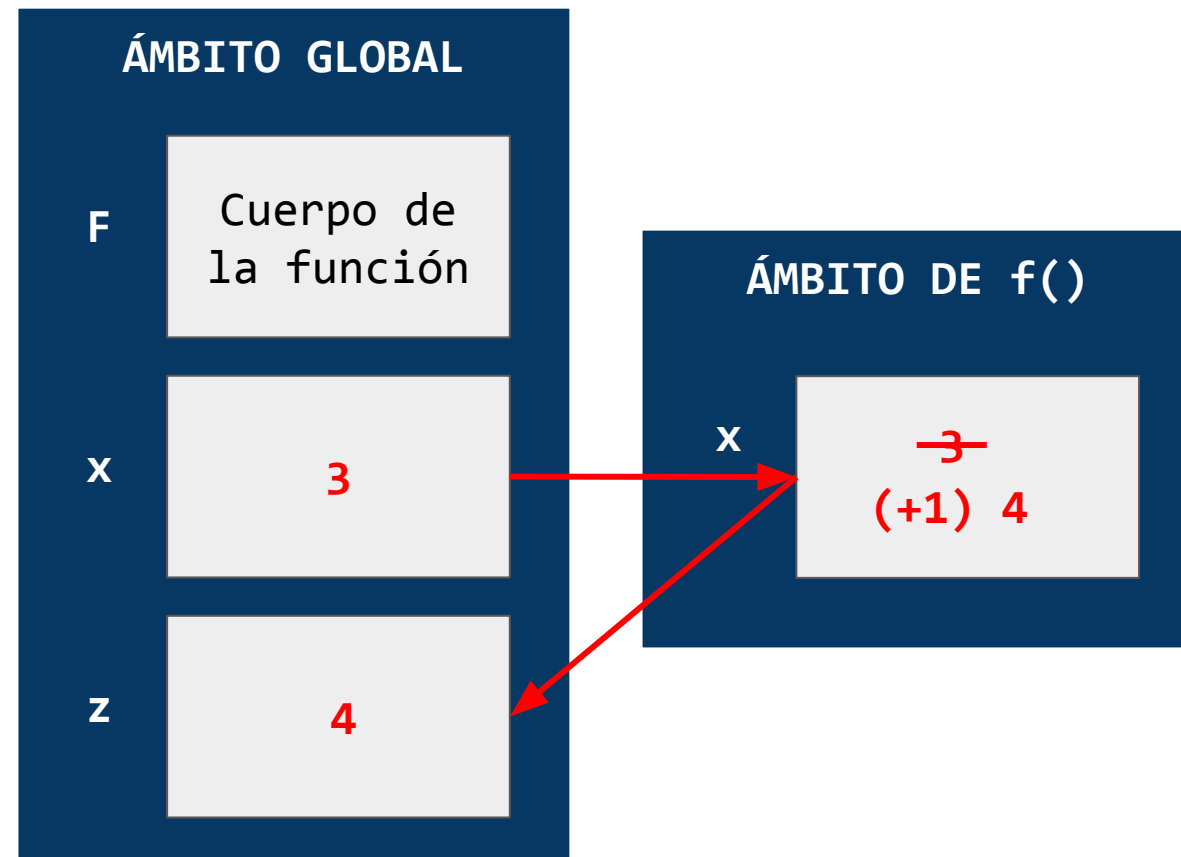
Ámbito de variables (I)

- 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] lista  
print("Antes:", valores)  
min, max = min_max(valores)  
print("Mínimo:", min, "; Máximo:", max)  
print("Después:"valores)
```

```
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") ← se usa el valor por defecto "s" de activo
```

```
autor("Stan", "Lee", "28/12/1922", "n") ← 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
```

Docstrings (I)

- Las **docstrings** nos proporcionan un mecanismo para asociar documentación a los módulos, funciones, clases y métodos
- Se añaden como comentarios, empleando **triple entrecomillado**, en la primera línea del elemento comentado
- A diferencia de los comentarios convencionales del código fuente, las *docstrings* describen **qué** hace la función, no **cómo**
- Son empleados por diferentes herramientas para la generación de documentación (*help*, *sphinx*), baterías de pruebas (*doctest*),...

```
>>> import math
>>> help(math.cos)
cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
```

Docstrings (II)

Guía de estilo

- Las líneas y párrafos **docstring** deben comenzar con letras mayúsculas y terminar con punto (.)
- La primera línea (**summary line**), que será una descripción breve, debe comenzar justo a continuación de la triple comilla
- Si hay más líneas, la segunda debe ser en blanco, separando la descripción breve del resto de la documentación
- Todas las funciones deberían tener un *docstring*
- El *docstring* de un módulo comenzará en la primera línea del archivo
- La triple comilla final se colocará al final de la primera línea (si no hay más) o sólo en una línea
- El **PEP** (*Python Enhancement Proposal*) **257** establece las convenciones para la creación de *docstrings*
- Ejemplo: https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html

Docstrings (III)

El módulo *doctest* y los *docstrings*

- El módulo *doctest* busca en los *docstrings* porciones de texto que simulan sesiones interactivas de Python con **ejemplos de uso** de las funciones y verifica que se ejecuten del modo propuesto.
- Esto nos va a permitir:
 - Verificar que los *docstrings* están **actualizados** y que los ejemplos interactivos funcionan tal cual se indica
 - Realizar **pruebas de regresión** (*regression tests*) desde los archivos de baterías de pruebas de nuestras funciones, verificando que siguen funcionando de forma correcta tras modificaciones
 - Escribir documentación de tipo **tutorial** para nuestros módulos y paquetes, ilustrándola con ejemplos de uso “*ejecutables*”

Docstrings (y IV)

```
"""Módulo de 'ejemplo'."""

def suma(*nums):
    """Devuelve la suma de los números de la lista.

    >>> suma(1, 2.5, -1.0)
    2.0

    Args:
        n (list): lista de números
    Returns:
        (float): sumatorio
    """
    suma = 0.0
    for num in nums:
        suma += num
    return suma
```

```
$ python3 -m doctest -v ejemplo.py
```

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 ← importamos el módulo 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))
```

```
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

02.04 - Python. Funciones

Módulos (V)

- Nuestro anterior programa podría quedar:

```
from fibo import fib, fib2
```

importamos las unciones fib y fib2 del
módulo **fibo**

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

```
# Imprime la serie de Fibonacci hasta num
```

```
fib(num)
```

invocamos la función sin utilizar el nombre del módulo

```
# Obtiene una lista con la serie de Fibonacci hasta num
```

```
print(fib2(num))
```

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
```

← creamos el *alias* **f** para el módulo **fibo**

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

```
# Imprime la serie de Fibonacci hasta num
```

```
f.fib(num)
```

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

```
# Obtiene una lista con la serie de Fibonacci hasta num
```

```
print(f.fib2(num))
```

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*) incluida 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

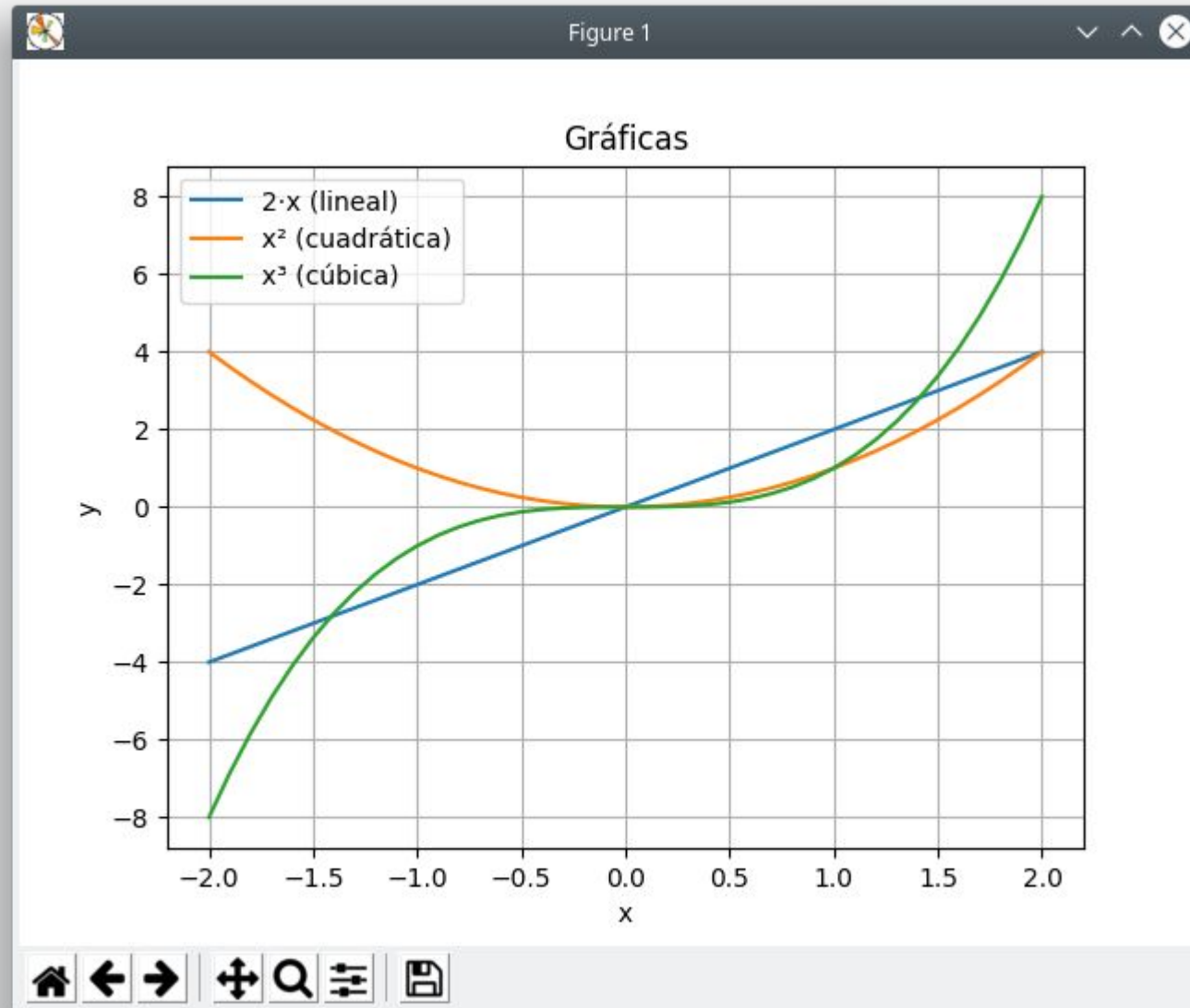
```
~$ pip3 install matplotlib
```

- Una vez instaladas, podremos importarlas mediante 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)



```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

02.05 - Python

Tipos estructurados

Introducción (I)

- Hasta el momento, todo el tratamiento computacional se ha realizado sobre datos de tipos primitivos (int, float, boolean) y un tipo particular de **colección**, las cadenas de texto (String)
- De igual modo que las funciones nos proporcionan un mecanismo para agrupar código en una única “entidad” e invocarlo cuando lo necesitemos, los lenguajes de programación suelen disponer de estructuras de datos que nos permiten agrupar **colecciones** de datos (de igual o diferente tipo) y tratarlos de modo conjunto.
- Python proporciona los siguientes tipos estructurados:

	ARRAY	TUPLA	LISTA	SET	DICCIONARIO
ORDENADO (ind)	X	X	X		
NO ORDENADO				X	X
MUTABLE	X		X	X (sin rep.)	X
INMUTABLE		X			

Introducción (II)

- Varios de los tipos estructurados para el manejo de colecciones son de tipo **secuencia** (array, lista, tupla, string). Cada elemento de la colección se encuentra en una posición concreta (índice) dentro de la secuencia.
- Operaciones comunes de secuencias (tipos mutables e inmutables):

Operación	Resultado
<code>x [not] in s</code>	True si <code>x</code> [no] es igual a un <i>item</i> de <code>s</code> . False en otro caso
<code>s + t</code>	Concatenación de las secuencias <code>s</code> y <code>t</code>
<code>s * n</code>	Equivalente a añadir <code>s</code> a sí mismo <code>n</code> veces
<code>s[i]</code>	Item en la posición <code>i</code> de la secuencia, con origen en 0
<code>s[i:j]</code>	Porción de la secuencia desde <code>i</code> hasta <code>(j-1)</code>
<code>s[i:j:k]</code>	Porción de la secuencia desde <code>i</code> hasta <code>(j-1)</code> con salto <code>k</code>
<code>len(s)</code>	Longitud de <code>s</code>
<code>min(s) max(s)</code>	Valores menor y mayor de <code>s</code>
<code>s.index(x[, i[, j]])</code>	Índice de la primera ocurrencia de <code>x</code> en <code>s</code> (desde <code>i</code> hasta <code>j</code>)(<code>i,j</code> no impl. siempre)
<code>s.count(x)</code>	Número total de ocurrencias del valor <code>x</code> en la secuencia <code>s</code>

Introducción (III)

- Operaciones comunes de secuencias de tipos **mutables**:

Operación	Resultado
<code>s[i] = x</code>	El <i>item</i> <code>i</code> es reemplazado por <code>x</code>
<code>s[i:j] = t</code>	La porción de <code>s</code> desde <code>i</code> hasta <code>j</code> es reemplazada por el contenido del iterable <code>t</code>
<code>del s[i]</code>	Elimina el <i>item</i> en la posición <code>i</code>
<code>del s[i:j]</code>	Elimina los <i>items</i> desde <code>i</code> hasta <code>(j-1)</code>
<code>del s[i:j:k]</code>	Elimina los <i>items</i> desde <code>i</code> hasta <code>(j-1)</code> y salto <code>k</code>
<code>s.append(x)</code>	Añade <code>x</code> al final de la secuencia <code>s</code>
<code>s.clear()</code>	Elimina todos los <i>items</i> de <code>s</code> (no disponible en array)
<code>s.copy()</code>	Crea una copia de <code>s</code>
<code>s.insert(i, x)</code>	Inserta <code>x</code> en la secuencia <code>s</code> en la posición de índice <code>i</code>
<code>s.pop()</code>	Elimina de la secuencia el último <i>item</i> . Devuelve el valor eliminado
<code>s.pop(i)</code>	Elimina de la secuencia el <i>item</i> en la posición <code>i</code> . Devuelve el valor eliminado
<code>s.remove(x)</code>	Elimina de la secuencia la primera ocurrencia de <code>x</code>
<code>s.reverse()</code>	Invierte el orden de la secuencia

Introducción (y IV)

- Una característica de estas estructuras de datos de tipo secuencia, al igual que otros objetos de tipo contenedor como los *String*, es que soportan la **iteración** a través de sus elementos.
- Esto nos permiten recorrerlos fácilmente mediante bucles *for*

```
>>> for element in [1, 2, 3]:  
...     print(element)  
>>> for element in (1, 2, 3):  
...     print(element)  
>>> for key in {'one':1, 'two':2}:  
...     print(key)  
>>> for char in {'123'}:  
...     print(char)  
>>> for line in open("myfile.txt"):  
...     print(line, end='')
```

Arrays (I)

- Los **arrays** son el método tradicional de crear **colecciones** de datos primitivos, donde todos los datos de la colección, son del **mismo tipo**. Si bien son populares en lenguajes como C, C++ o Java, no lo son tanto en Python, pues dispone de tipos estructurados más flexibles (listas).
- En general, cuando la gente habla de arrays en Python, suelen referirse a las listas. Sin embargo, son estructuras de datos diferentes. En Python, los arrays pueden ser vistos como una manera **eficiente** de almacenar cierta listas, donde todos los elementos son del mismo tipo.
- En Python, el soporte de arrays lo proporciona el **módulo array** que debe ser **importado** antes de poder inicializarlos y usarlos.
- Durante la **creación** del array, deberemos indicar su **tipo** de datos, **limitando** de esta manera el **rango** y tipo de los valores almacenados
- Documentación oficial: <https://docs.python.org/3/library/array.html>

Arrays (II)

CÓDIGO TIPO	TIPO C EQUIV	TIPO PYTHON	TAMAÑO(Bytes)	RANGO VALORES
'b'	signed char	int	1	$[-2^7, 2^7-1]$ [-128, 127]
'B'	unsigned char	int	1	$[0, 2^8-1]$ [0, 255]
'u'	wchar_t	Unicode	2	'\u0000' - '\uFFFF'
'h'	signed short	int	2	$[-2^{15}, 2^{15}-1]$ [-32.768, 32.767]
'H'	unsigned short	int	2	$[0, 2^{16}-1]$ [-65.536, 65.535]
'i'	signed int	int	4(2*)	$[-2^{31}, 2^{31}-1]$ [-2.147.483.648, 2.147.483.647]
'I'	unsigned int	int	4(2*)	$[0, 2^{32}-1]$ [0, 4.294.967.296]
'l'	signed long	int	8(4*)	$[-2^{63}, 2^{63}-1]$ [-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]
'L'	unsigned long	int	8(4*)	$[0, 2^{64}-1]$ [0, 18.446.744.073.709.551.615]
'f'	float	float	4	[1.2E-38, 3.4E+38]
'd'	double	float	8	[2.3E-308, 1.7E+308]

Tabla de *códigos de tipo* para la creación de arrays en Python (*dep. arquitectura)

Arrays (y III). Ejemplos

```
>>> import array as arr
>>> udata = arr.array('B', (0, 18, 128))
>>> udata
array('B', [0, 18, 128])
>>> udata.append(255)
>>> udata
array('B', [0, 18, 128, 255])
>>> udata.append(300)
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    udata.append(300)
OverflowError: unsigned byte integer is greater than maximum
>>> udata[1]
18
>>> chars = arr.array('u', 'hola')
>>> udata.append('\u0061')
>>> chars
array('u', 'holaa')
>>> chars.count('a')
2
```

creación de un array de bytes

adición de elementos al array

error al tratar de añadir valores fuera del rango permitido

count() devuelve el número de ocurrencias en el array del *item* indicado

Tuplas (I)

- Las **tuplas** son secuencias **inmutables**, generalmente empleadas para el almacenamiento **ordenado** de datos **heterogéneos**.
- Su diferencia fundamental con las **listas** es que, una vez creada, no puede ser modificada, a diferencia de las listas. Suelen emplearse para almacenar secuencias de valores constantes (por ejemplo, las claves de un diccionario) o en retornos de funciones de varios valores.
- Las tuplas pueden ser construidas de diferentes maneras:
 - Usando un par de **paréntesis** para denotar la **tupla vacía**: `()`
 - Usando una **coma final** para una tupla simple: `a`, ó `(a,)`
 - Separando los *items* con **comas**: `a, b, c` ó `(a, b, c)`
 - Usando el constructor **tuple()**: `tuple()` ó `tuple(iterable)`
- En realidad es la coma, no el paréntesis, lo que determina a una tupla. Son opcionales salvo en casos ambiguos (paso parámetros a función,...)

Tuplas (II). Ejemplos

```
>>> t = (2, 'uno', 3)
```

← creación de una **tupla**

```
>>> t
```

```
(2, 'uno', 3)
```

```
>>> t[0]
```

← acceso a los elementos mediante **índice**

```
2
```

```
>>> (2, 'uno', 3) + (5, 6)
```

← creación de una nueva tupla por **concatenación**

```
(2, 'uno', 3, 5, 6)
```

```
>>> t[1:3]
```

← extracción de **segmentos** (tuplas) de la tupla

```
('uno', 3)
```

```
>>> type(t[1:3])
```

```
<class 'tuple'>
```

```
>>> t[2:3]
```

```
(3,)
```

```
>>> type(t[2:3])
```

```
<class 'tuple'>
```

```
>>> t[1] = 4
```

← error al tratar de **modificar** un elemento de la tupla

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

```
  File "<pyshell#10>", line 1, in <module>
```

```
    t[1] = 4
```

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


Tuplas (y III). Ejemplos

- Las tuplas se emplean para **intercambiar** el valor de variables:

```
>>> x = 5
>>> y = 6
>>> x = y
>>> y = x
>>> x
6
>>> y
6
```



```
>>> x = 5
>>> y = 6
>>> temp = y
>>> y = x
>>> x = temp
>>> x
6
>>> y
5
```



```
>>> x = 5
>>> y = 6
>>> x, y = y, x
>>> x
6
>>> y
5
```



- Para **retornar más de un valor** desde una función:

```
def div_entera ( x, y ):
    c == x // y
    r == x % y
    return c, r
```

c,r es una tupla

cociente, resto
es una tupla

```
cociente, resto = div_entera(5, 4)
```

Listas (I)

- Como las tuplas, las **listas** son secuencias ordenadas de datos del mismo o diferente tipo. La principal diferencia es que las listas son **mutables**. Es decir, una vez creadas, pueden ser modificadas.
- Este hecho aporta una gran flexibilidad. Por otro lado, puede dar lugar a situaciones indeseables. Por ejemplo, una lista pasada a una función, podría ver modificado su contenido en el interior de la misma.
- Las listas pueden ser construidas de diferentes maneras:
 - Usando un par de **corchetes** para denotar la **lista vacía**: `[]`
 - Usando **corchetes**, separando los *items* con comas: `[a]`, ó `[a, b, c]`
 - Usando listas **por comprensión**: `[x for x in iterable]`
 - Usando el constructor **list()**: `list()` ó `list(iterable)`
- Además de los métodos y operaciones de las secuencias mutables, las listas disponen también del método **sort()**

Listas (II). Indexado

- Como el resto de secuencias ordenadas, podemos acceder a los elementos de la lista mediante un **índice**:

```
>>> lista_1 = []  
>>> lista_2 = [2, 'a', 4, True]  
>>> lista_2  
[2, 'a', 4, True]  
>>> len(lista_1)  
0  
>>> lista_2[0]  
2  
>>> lista_2[2] + 1  
5  
>>> lista[7]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: list index out of range  
>>> i = 2  
>>> lista_2[i-1]  
'a'
```

← creación de listas

← obtención de la longitud de la lista

← acceso a los elementos mediante índice

← error al tratar de acceder fuera del rango del índice de la lista

Listas (II). Ejemplos

- Disponemos de diferentes mecanismos para **añadir**, **modificar** y **eliminar** elementos de la lista:

```
>>> lista_1 = [1, 2]
>>> lista_2 = lista_1 + [2, 'a', 4]
>>> lista_2
```

creación de una nueva
lista por **concatenación**

```
[1, 2, 2, 'a', 4]
>>> lista_2.append(lista_1)
```

adición de elementos a la lista

```
>>> lista_2
[1, 2, 2, 'a', 4, [1, 2]]
>>> len(lista_2)
```

```
6
>>> lista_2[3] = 9
>>> lista_2[5][1] = 4
```

acceso y **modificación** de los elementos
de la lista mediante **índice**

```
>>> lista_2
[1, 2, 2, 9, 4, [1, 4]]
>>> lista_2.sort()
```

error por tratar de ordenar una lista
de elementos **heterogénea**

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'list' and 'int'
```

Listas (III). Métodos *append()* y *extend()*

- Diferencia entre los métodos **append()** y **extend()**

```
>>> lista = [1, 2]
>>> num = 5
>>> tupla = (3, 4)
>>> cadena = 'hola'
>>> lista.append(num)
>>> lista.append(tupla)
>>> lista.append(cadena)
>>> lista
[1, 2, 5, (3, 4), 'hola']
```

append() añade los nuevos elementos, manteniendo su **tipo**, al final de la lista

```
>>> lista = [1, 2]
>>> lista.extend(tupla)
>>> lista.extend(cadena)
>>> lista
[1, 2, 3, 4, 'h', 'o', 'l', 'a']
```

extend() añade de forma individual cada uno de los elementos del **iterable**

```
>>> lista.extend(num)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

error por tratar de **extender** la lista con un objeto no **iterable**

Listas (IV). Métodos `del()`, `remove()` y `pop()`

- Diferencia entre los métodos `del()`, `remove()` y `pop()`

```
>>> lista = [x for x in range(1, 11)]
```

← creación de una lista por comprensión

```
>>> lista
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> del lista[2]
```

```
>>> lista
```

```
[1, 2, 4, 5, 6, 7, 8, 9, 10]
```

← `del()` permite eliminar *items* concretos o **segmentos** enteros identificados mediante **índices**

```
>>> del lista[6:]
```

```
>>> lista
```

```
[1, 2, 4, 5, 6, 7]
```

```
>>> lista.pop()
```

```
7
```

← `pop()` elimina el **último** *item* de la lista o el correspondiente al **índice** indicado. **Devuelve** el valor eliminado

```
>>> lista.pop(1)
```

```
2
```

```
>>> lista
```

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

```
>>> lista.remove(4)
```

```
>>> lista
```

← `remove()` elimina de la lista la **primera** ocurrencia en la lista del **valor** indicado. Genera un **error** si valor no se encuentra en la lista

```
[1, 5, 6]
```

Listas (V). Conversión Listas-Strings

- Conversiones entre listas y cadenas de caracteres:
 - *list(s)*, crea una lista a partir de cada carácter de la cadena *s*
 - *s.split(sep)*, trocea la cadena *s* en subcadenas utilizando *sep* como separador (espacio si no se indica) y genera una lista
 - *s.join(lista)* genera una cadena de caracteres concatenando los elementos (tipo *str*) de *lista* con la cadena *s*

```
>>> s = "hola qué tal?"
>>> list(s)
['h', 'o', 'l', 'a', ' ', 'q', 'u', 'é', ' ', 't', 'a', 'l', '?']
>>> s.split()
['hola', 'qué', 'tal?']
>>> list(s.split()[0])
['h', 'o', 'l', 'a']
>>> s.split('a')
['hol', ' ', 'qué t', 'l?']
>>> '/' .join(['15', '02', '2018'])
'15/02/2018'
```

Listas (V). Ordenación

- Ordenación de listas. Los elementos deben ser del **mismo tipo** primitivo para poder ordenarlos con los operadores < >
 - El método **sort()** ordena la lista (la modifica)
 - La función **sorted(lista)**, devuelve una lista ordenada a partir de **lista** (no la modifica)
 - El método **reverse()** invierte el orden de la lista (la modifica)

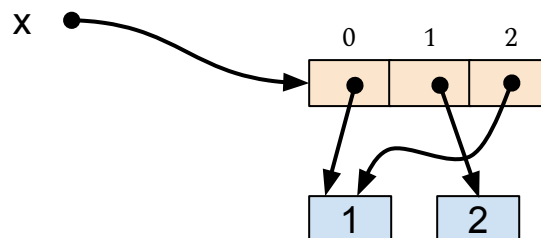
```
>>> lista = [3, 2, 1, 8, 2]
>>> sorted(lista)
[1, 2, 2, 3, 8]
>>> lista
[3, 2, 1, 8, 2]
>>> lista.sort()
>>> lista
[1, 2, 2, 3, 8]
>>> lista.reverse()
>>> lista
[8, 3, 2, 2, 1]
```


Listas (VI). Mutabilidad y Clonado

- Las listas son **objetos mutables** y esto puede tener **efectos colaterales** cuando trabajamos con ellas
- Las listas se implementan como **objetos en memoria** y las variables de tipo lista contienen **referencias** que apuntan a dichos objetos
- Cuando asignamos una variable de tipo lista a otra variable, no se realiza una copia de dicha listas, sino que ambas contendrán la misma referencia al mismo objeto en memoria (son **alias** de la misma lista).
- De lo anterior se desprende que, si **modificamos** una de las listas anteriores, la otra también se verá modificada. En realidad, hay una única lista a la que apuntan ambas variables
- Esto mismo ocurre cuando pasamos una lista como argumento a una función. No se pasa una copia de la lista, sino una referencia a la misma.

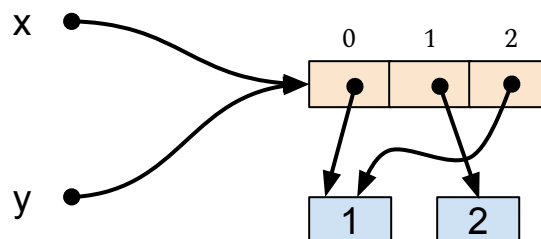
Listas (VII). Mutabilidad y Clonado

`x = [1, 2, 1]`



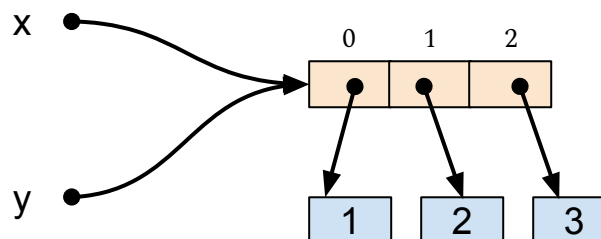
`x[2] → 1`

`y = x`



`x[2] → 1`
`y[2] → 1`

`x[2] = 3`



`x[2] → 3`
`y[2] → 3`

Listas (VIII). Mutabilidad y Clonado

- Cuando lo que queremos es obtener una copia de una lista, podemos:
 - Crear una función que recorra la lista y devuelva una lista con la copia de los valores

```
def copia_lista (lista):  
    copia = []  
    for item in lista: copia += [item]  
    return copia
```

- Emplear el método *copy()* para obtener una nueva lista

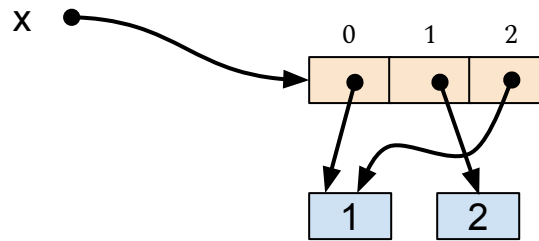
```
>>> lista = [1, 2, 3]  
>>> copia = lista.copy()
```

- Emplear el *operador de indexado* para obtener una nueva lista

```
>>> lista = [1, 2, 3]  
>>> copia = lista[:]
```

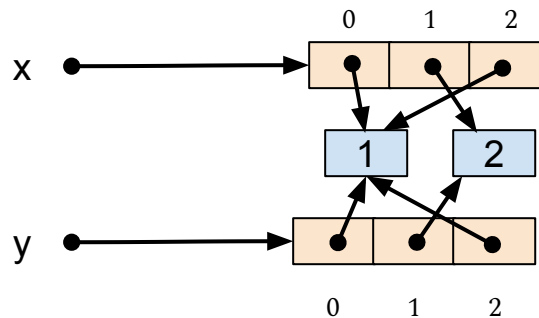
Listas (IX). Mutabilidad y Clonado

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



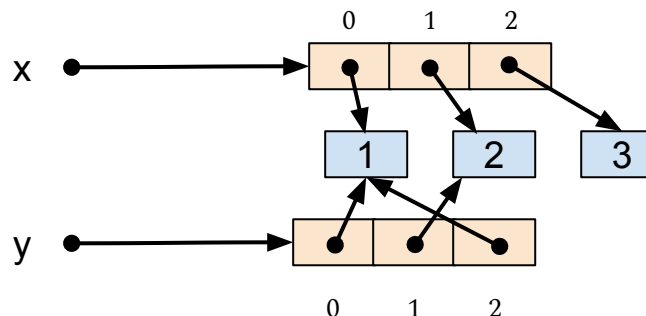
$x[2] \rightarrow 1$

```
y = x[:]
```



$x[2] \rightarrow 1$
 $y[2] \rightarrow 1$

```
x[2] = 3
```



$x[2] \rightarrow 3$
 $y[2] \rightarrow 1$

Listas (X). *Swallow vs Deep copy*

- Los ejemplos de copia de listas vistos hasta el momento son ejemplos de lo que se denomina *swallow copy* (copia superficial). Dado que las listas pueden almacenar cualquier tipo de dato, podríamos definir listas contenidas en otras listas. ¿Qué ocurre al modificarlas?

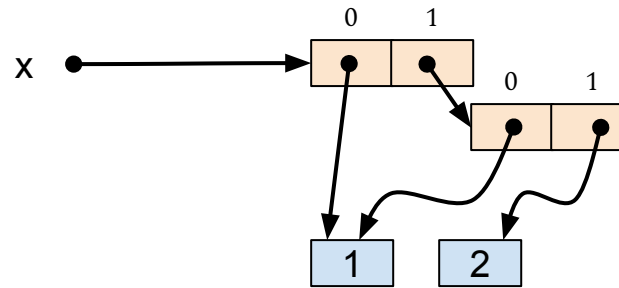
```
>>> colores = [['azul', 'verde'], ['rojo', 'naranja']]
>>> colores_copia = colores[:]
>>> colores_copia
[['azul', 'verde'], ['rojo', 'naranja']]
>>> del colores[0]
>>> colores
[['rojo', 'naranja']]
>>> colores_copia
[['azul', 'verde'], ['rojo', 'naranja']]
>>> del colores_copia[1][0]
>>> colores_copia
[['azul', 'verde'], ['naranja']]
>>> colores
[['naranja']]
```



¡¡ SE MODIFICA
LA LISTA
ORIGINAL !!

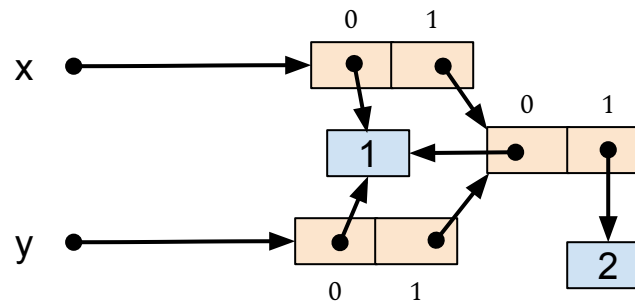
Listas (XI). *Swallow vs Deep copy*

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



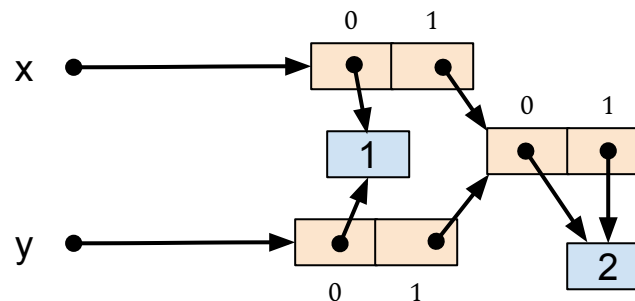
```
x[1] → [1, 2]
```

```
y = x[:]
```



```
x[1] → [1, 2]
y[1] → [1, 2]
```

```
x[1][0] = 2
```




```
x[1] → [2, 2]
y[1] → [2, 2]
```

swallow copy (copia superficial)

Listas (XI). *Swallow vs Deep copy*

- Cuando tratamos con objetos compuestos, Python nos proporciona funciones para realizar una *deep copy* (copia profunda). Esto es, se construye un nuevo objeto compuesto y, recursivamente, se insertan en él copias de los objetos encontrados en el original

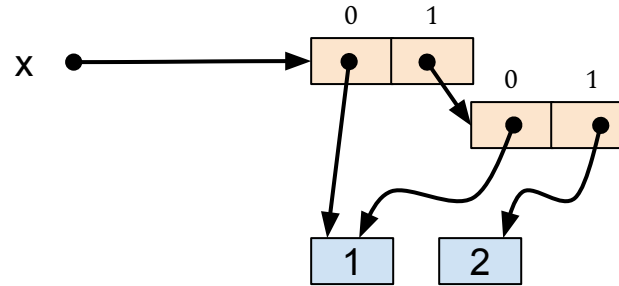
```
>>> import copy
>>> colores = [['azul', 'verde'], ['rojo', 'naranja']]
>>> colores_copia = copy.deepcopy(colores)
>>> colores_copia
[['azul', 'verde'], ['rojo', 'naranja']]
>>> del colores[0]
>>> colores
[['rojo', 'naranja']]
>>> colores_copia
[['azul', 'verde'], ['rojo', 'naranja']]
>>> del colores_copia[1][0]
>>> colores_copia
[['azul', 'verde'], ['naranja']]
>>> colores
[['rojo', 'naranja']]
```



LOS CAMBIOS
DE UNA LISTA
NO AFECTAN A
LA OTRA

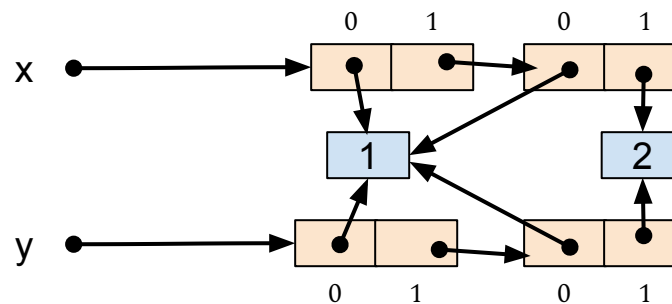
Listas (XII). *Swallow vs Deep copy*

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



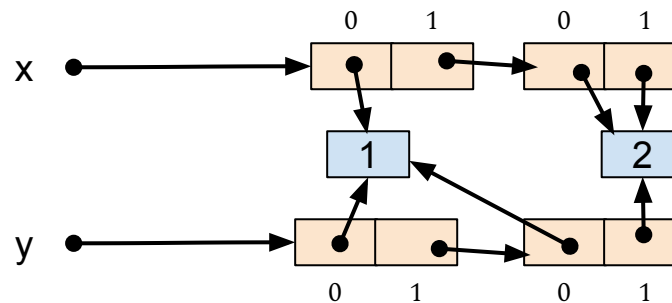
```
x[1] → [1, 2]
```

```
y = copy.deepcopy(x)
```



```
x[1] → [1, 2]
y[1] → [1, 2]
```

```
x[1][0] = 2
```



```
x[1] → [2, 2]
y[1] → [1, 2]
```

deep copy (copia profunda)

Sets (I)

- Los **sets** o **conjuntos**, son secuencias de elementos que cumplen:
 - **No están ordenados**. No soporta indexado.
 - Cada elemento es **único** (no se permiten duplicados)
 - El set es **mutable** pero sus elementos tienen que ser de tipo **inmutable**
- Usos habituales de los sets son su empleo como claves de diccionarios, eliminación de duplicados y el cálculo de operaciones matemáticas del álgebra de conjuntos (unión, intersección, diferencia, complemento,...)
- Al igual que el resto de colecciones, soporta el operador pertenencia (**in**), la función **len()** y su recorrido mediante bucles **for**.
- Los sets pueden ser contruidos de diferentes maneras:
 - Usando **llaves**, separando los *items* con **comas**: **{1}**, ó **{a, b, c}**
 - Usando el constructor **set()**: **set()** ó **set(iterable)**

Sets (II). Ejemplos

```
>>> set_1 = {}
```

```
>>> set_2 = {2, 'a', (1, 4), True}
```

```
>>> set_3 = set('hola')
```

```
>>> set_2
```

```
{2, 'a', (1, 4), True}
```

```
>>> set_3
```

```
{'h', 'o', 'l', 'a'}
```

```
>>> len(set_1)
```

```
4
```

```
>>> s1.add(3)
```

```
>>> s1.add('hola')
```

```
>>> s1.add(3)
```

```
>>> s1
```

```
{'hola', 3}
```

```
>>> s1.pop()
```

```
'hola'
```

```
>>> s1.discard(3)
```

```
>>> s1.remove(3)
```

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

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

```
KeyError: 3
```

creación de sets

obtención de la longitud del set

añadir elementos al set mediante add

el orden puede ser diferente al que fueron añadidos
y no se añaden duplicados de elementos ya existentes

borrado de elementos del set mediante pop, discard y remove

Si el elemento a borrar no existe, **remove**
genera una **excepción** de tipo **KeyError**.
discard no genera excepción si no existe

Sets (III)

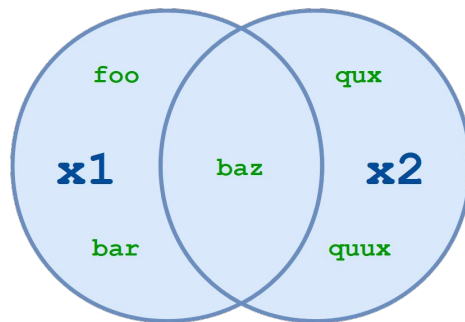
- Operaciones de álgebra de conjuntos:

Operación	Ejemplo
<code>x1.union(x2[, x3 ...])</code> <code>x1 x2 [x3 ...]</code>	<code>>>> {1, 2, 3} {3, 6} {8}</code> <code>{1, 2, 3, 6, 8}</code>
<code>x1.intersection(x2[, x3 ...])</code> <code>x1 & x2 [& x3 ...]</code>	<code>>>> {1, 2, 3} & {3, 6}</code> <code>{3}</code>
<code>x1.difference(x2[, x3 ...])</code> <code>x1 - x2 [- x3 ...]</code>	<code>>>> {1, 2, 3} - {3, 6}</code> <code>{1, 2}</code>
<code>x1.symmetric_difference(x2)</code> <code>x1 ^ x2 [^ x3 ...]</code>	<code>>>> {1, 2, 3} ^ {3, 6}</code> <code>{1, 2, 6}</code>
<code>x1.isdisjoint(x2)</code>	<code>>>> {1, 2, 3}.isdisjoint({3, 6})</code> <code>False</code>
<code>x1.issubset(x2)</code> <code>x1 <= x2</code>	<code>>>> {2, 1} <= {1, 2, 3}</code> <code>True</code>
<code>x1 < x2</code>	<code>>>> {2, 1} < {1, 2, 3}</code> <code>True</code>
<code>x1.issuperset(x2)</code> <code>x1 >= x2</code>	<code>>>> {2, 1, 3} >= {1, 2}</code> <code>True</code>
<code>x1 > x2</code>	<code>>>> {2, 1, 3} > {1, 2}</code> <code>True</code>

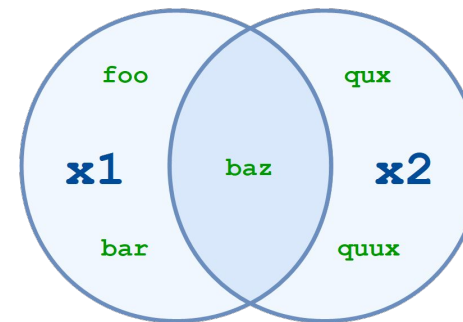
Sets (y IV)

```
x1 = {'foo', 'bar', 'baz'}  
x2 = {'baz', 'qux', 'quux'}
```

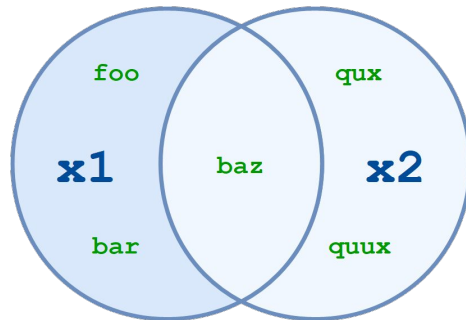
`x1.union(x2)`
`x1 | x2`



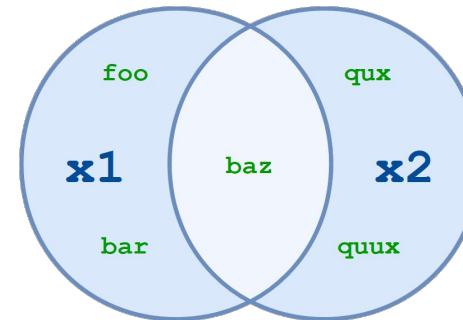
`x1.intersection(x2)`
`x1 & x2`



`x1.difference(x2)`
`x1 - x2`



`x1.symmetric_difference(x2)`
`x1 ^ x2`



Diccionarios (I)

- Los **diccionarios** son estructuras de datos del tipo **arrays asociativos**. Consisten en colecciones **no ordenadas** y **mutables** de **pares *clave:valor***, donde cada clave referencia (*indexa*) un valor concreto. El acceso a los diferentes valores almacenados en el diccionario se realiza a través de la clave correspondiente.
- Las agendas o los diccionarios de las lenguas, son típicos ejemplos de diccionarios. También lo sería cualquier tabla cuyos registros estuvieran identificados por un campo clave, índice o ***primary key***
- Las claves son **únicas** (no puede haber claves duplicadas) y deben ser de un tipo **inmutable** (tipos primitivos y tuplas, si no contienen tipos mutables)
- No hay restricciones en cuanto al tipo de datos de los valores. Además, distintas claves pueden tener valores de diferentes tipos

Diccionarios (II)

- Los diccionarios pueden ser contruidos de diferentes formas:
 - Usando **llaves** para denotar un diccionario vacío: `{}`
 - Usando llaves, separando las parejas **clave:valor** mediante comas:
`{ 1:'uno', 2:'dos', 'otra_clave':'otro_valor' }`
 - Usando el constructor: **`dict()`** ó **`dict(secuencia de parejas)`**
- Para añadir nuevas parejas **clave:valor** al diccionario, utilizaremos:
`diccionario[nueva_clave] = nuevo_valor`
Si la clave ya existía, se actualizará su valor al nuevo valor proporcionado
- Para eliminar entradas del diccionario, podemos usar la función **`del()`** o el método **`pop()`**, indicando la clave del elemento a eliminar. Ambas generan una excepción **`KeyError`** si la clave no existe.
- El método **`clear()`** elimina todas las entradas de un diccionario

Diccionarios (III). Ejemplos

```
>>> dict_1 = {}
>>> dict_2 = {1:'uno', 2:'dos', 'a':(1, 2, 3)}
>>> dict_2
{1:'uno', 2:'dos', 'a':(1, 2, 3)}
>>> dict_3 = dict([('G':6.6742e-11), ('pi':3.1416)])
>>> dict_2
{1:'uno', 2:'dos', 'a':(1, 2, 3)}
>>> dict_3['c']=299792458
>>> dict_3
{'G':6.6742e-11, 'pi':3.1416, 'c':299792458}
>>> dict_2['a']
(1, 2, 3)
>>> dict_2['a'][2]
3
>>> dict_2['a']='letra_a'
>>> dict_2.get('a')
'letra_a'
>>> del dict_2['a']
>>> dict_2.pop(1)
>>> dict_2
{2:'dos'}
```

creación de
diccionarios

adición de elementos al diccionario

acceso y modificación de los elementos

eliminación de elementos

Diccionarios (IV). Unión de diccionarios

- El método `update()` permite fusionar los contenidos de dos diccionarios. Al invocar `d1.update(d2)`, el diccionario `d1` incorporará todas las entradas del diccionario `d2`. Si alguna de las claves de `d2` ya se encuentra en `d1`, ésta se actualizará con el valor de `d2`

```
>>> d1 = {1:'uno', 2:'dos', 3:'tressss'}
>>> d2 = {4:'cuatro', 3:'tres'}
>>> d1.update(d2)
>>> d1
{1: 'uno', 2: 'dos', 3: 'tres', 4: 'cuatro'}
```

- Desde Python 3.5, se dispone de una nueva sintaxis (operador `**`) para generar **nuevos** diccionarios a partir de la **unión** de otros (PEP 448)

```
>>> d1 = {1:'uno', 2:'dos', 3:'tressss'}
>>> d2 = {4:'cuatro', 3:'tres'}
>>> d3 = {**d1, **d2, 5:'cinco', 6:'seis'}
>>> d3
{1: 'uno', 2: 'dos', 3: 'tres', 4: 'cuatro', 5: 'cinco', 6: 'seis'}
```


Diccionarios (V). Vistas

- Los métodos *items()*, *keys()* y *values()* nos proporcionan objetos tipo vista muy útiles a la hora de recorrer y acceder a los valores de los diccionarios
- *items()* devuelve las parejas *clave:valor* contenidas en el diccionario

```
>>> d = {1:'uno', 2:'dos', 3:'tres'}
>>> d.items()
dict_items([(1, 'uno'), (2, 'dos'), (3, 'tres')])
>>> list(d.items())
[(1, 'uno'), (2, 'dos'), (3, 'tres')]
>>> list(d.items())[1][1]
'dos'
>>> for k,v in d.items():
...     print(k,v)
1 uno
2 dos
3 tres
```

Diccionarios (y VI). Vistas

- *keys()* devuelve las **claves** contenidas en el diccionario
- *values()* devuelve una vista de los **valores** en el diccionario

```
>>> d = {1:'uno', 2:'dos', 3:'tres'}
>>> d.keys()
dict_keys([1, 2, 3])
>>> 4 in d.keys()
False
>>> d.values()
dict_values(['uno', 'dos', 'tres'])
>>> list(d.values())
['uno', 'dos', 'tres']
>>> 'uno' in d.values()
True
>>> for k in d.keys():
...     print(k, d[k])
1 uno
2 dos
3 tres
```

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: Could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.client=ho
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

P R O G R A M A C I Ó N

03.02 - P00

Python

Objetos en Python (I)

- Ya sabemos que Python soporta diferentes tipos de datos:

1234 3.14159 "Hola" [1, 2, 3, 5, 7, 11]
{ "GZ": "Galicia", "MD": "Madrid", "AS": "Asturias" }

- Cada uno de estos **objetos** o **instancias** se caracterizan por:
 - un **tipo** (clase)
 - una **representación interna de datos** (mediante tipos primitivos o por composición de objetos)
 - un conjunto de procedimientos para **interactuar** con el objeto
- Cada **instancia** es un tipo particular de objeto:
 - 1234 es una instancia de un *int* (**class 'int'**)
 - "Hola" es una instancia de un *string* (**class 'str'**)
 - 3.14159 es una instancia de un *float* (**class 'float'**)

Objetos en Python (II)

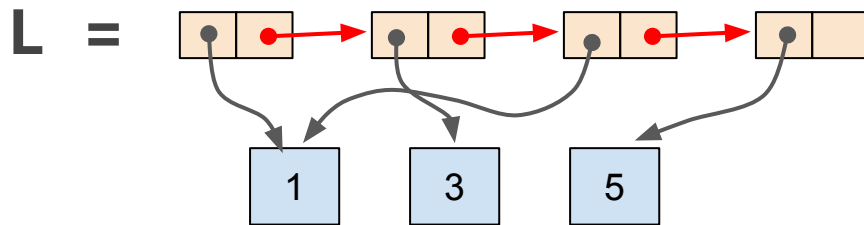
- En Python **todo** son objetos pertenecientes a un tipo (clase) determinado
- Estos objetos son **abstracciones** que encapsulan:
 - una **representación interna** mediante atributos de datos
 - un **interfaz** para interactuar con los objetos a través de métodos (procedimientos). Define el comportamiento pero oculta la implementación
- Se pueden **crear nuevas instancias** u objetos
- Se pueden **destruir objetos**:
 - Explícitamente, usando el comando **del**, o simplemente “olvidándonos” de ellos
 - Python dispone de un **“garbage collector”** (recolector de basura) que “eliminará” definitivamente aquellos objetos destruidos o inaccesibles (objetos cuya cuenta de referencias está a 0)

Objetos en Python (y III)

- Veamos un caso particular:

`[1, 3, 1, 5]` es un objeto de tipo **lista** (class 'list')

- ¿Cuál es su **representación interna**?



listas enlazadas
(realmente, nos da igual)

- ¿Cómo **manipulamos** la lista?

- `L[i]`, `L[i:j]`, `L[i,j,k]`, `+`
- `len()`, `min()`, `max()`, `del(L[i])`
- `L.append()`, `L.extend()`, `L.count()`, `L.index()`, `L.insert()`, `L.pop()`, `L.remove()`, `L.reverse()`, `L.sort()`

- La representación interna debe ser **privada** (oculta). Se interactúa con el objeto a través de **interfaces definidos**

Definición de Clases (I)

- En general, la creación de una nueva clase supone:
 - definir el **nombre** de la clase
 - definir sus **atributos**
 - definir e implementar sus **métodos**
- Usaremos la palabra reservada **class** para definir una nueva clase:

```
      nombre clase   clase padre
class Coordinada(object):
    <definición de atributos y métodos>
```

- De igual modo al resto de estructuras de Python, tendremos que indentar el código para delimitar aquel que pertenece a la definición de la clase
- **object** indica que la clase deriva de ella. En Python 3 todas las clases derivan por defecto de object (se puede omitir de la definición). Es decir, object es una **superclase** de todas las clases en Python 3

Definición de Clases (II)

- Para poder crear objetos de la clase, necesitamos que la clase defina un método especial denominado **constructor**. Este método se invoca al crear el nuevo objeto y, a través de él, podemos pasarle argumentos a la clase para parametrizar la creación del objeto.
- A través de la clase **object**, Python proporciona un método constructor a todas las clases, llamado **__init__**, que nosotros podremos sobrescribir.
- Normalmente, los atributos de la clase se definen en el método **__init__** y se les asignan sus valores iniciales.
- Todos los métodos de las clases en Python, incluido **__init__**, tienen como primer argumento una variable denominada (por convención) **self**. Esta variable contiene una referencia al propio objeto y se emplea para poder “*identificar*” los atributos propios del objeto y diferenciarlos de cualquier otra variable local del mismo nombre.

Definición de Clases (y III)

- Siguiendo con el ejemplo anterior:

```
class Coordenada:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

Coordenada
- x
- y

método
constructor
heredado
de la clase
object

datos de **inicialización** de
estado del nuevo objeto

parámetro con la **referencia** a
la nueva instancia

- La clase define dos nuevos **atributos**, de nombres **x** e **y**
- Estos atributos se inicializan a los **argumentos** **x** e **y** del constructor
- El uso de **self** permite distinguir entre los atributos del objeto y los argumentos del constructor

Creando objetos

- Vamos a crear objetos de la clase Coordenada anterior:

```
c = Coordenada(3, 4)
origen = Coordenada(0, 0)
print(c.x)
print(origen.x, origen.y)
```

Creamos nuevos objetos (instancias) de la clase `Coordenada` utilizando su nombre de clase

Proporcionamos los parámetros necesarios según la definición del constructor. El argumento para `self` no se proporciona, es proporcionado automáticamente por Python

- Empleamos el **operador `"."`** para acceder a los miembros del objeto
- **`c.x`** se interpreta como *"accede al valor asociado a `x` en el contexto (frame) `c`", es decir, la **variable de instancia `x`** del **objeto `c`***

La función **`isinstance(obj, class)`** nos permite chequear si un objeto concreto es una instancia de una clase determinada. Ej:

```
isinstance(c, Coordenada)
```

Añadiendo métodos (I)

- Con la salvedad de que sólo actúan dentro de la clase, los métodos son equivalentes a las funciones: reciben parámetros, realizan operaciones devuelven valores.
- Python siempre pasa una referencia del objeto como primer argumento, por lo que todos los métodos deberán tener como mínimo un parámetro. Por convención, el nombre de ese parámetro es *self*
- Al igual que con los atributos, emplearemos el **operador “.”** para acceder a los diferentes métodos proporcionados por la clase
- Una clase heredará todos los métodos de su *superclase*. Opcionalmente, podrá proporcionar su propia definición de dichos métodos de forma que adapte su comportamiento a sus necesidades (*overriding*)
- Cualquier método puede ser invocado como **método de clase** (eq. Java *static*). Recuerda que debe recibir, al menos, un argumento

Añadiendo métodos (II). Métodos getter/setter

- Con objeto de garantizar la **encapsulación**, no deberíamos poder acceder directamente a la **estructura interna** de los objetos sino a través del **interfaz** correspondiente. Así, el diseñador es libre de modificarla.
- Todas las variables miembro deberían permanecer **privadas**, es decir, ocultas al exterior. Los lenguajes POO suelen incluir modificadores para configurar el acceso a las variables (*public*, *private*, *protected*)
- Para aquellas que precisen ser “accedidas” para leer o modificar su valor, la clase debe proporcionar los métodos de interfaz correspondientes
- Tradicionalmente, estos métodos reciben el nombre de **getters** (lectura) y **setters** (modificación). Su nombre se forma a partir de los prefijos **get** o **set**, y del nombre de la variable miembro. Por ej: **get_var()** o **getVar()**
- Python no dispone de modificadores de acceso para las variables. La ocultación de las variables miembro y la definición de sus métodos **get/set** se implementa mediante la definición de **propiedades**

Añadiendo métodos (III). Métodos getter/setter

- Continuemos con el ejemplo de la clase Coordenada. Vamos a añadir los *getter* y *setter* para modificar los valores *x* e *y* de los objetos.
- Vamos a implementarlos del modo tradicional (Java, C++,...). Más adelante veremos el *"pythonic way"* de hacerlo, sin duda, más “elegante”

```
class Coordenada:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def get_x(self):
        return self.x
    def set_x(self, x):
        self.x = x
    def get_y(self):
        return self.y
    def set_y(self, y):
        self.y = y
```

Coordenada
- x - y
get_x() set_x() get_y() set_y()

Fíjate que todos los métodos incluyen el argumento *self* como primer parámetro para poder acceder a las variables miembro de la instancia

Añadiendo métodos (IV). Métodos getter/setter

- La definición de *getters* y *setters* independendiza la estructura de datos interna del acceso al estado de los objetos. De este modo se pueden hacer cambios en la estructura (cambio de nombres de variables, de tipos,...) sin afectar a los programas existentes que usen dichos objetos
- Debemos tener cuidado en Python ya que, al no disponer de modificadores de acceso, podremos acceder directamente a dichas variables miembro

```
c = Coordinada(3, 4)
c.x = 5           # set
print(c.x)        # get
```
- En Python se emplean diversas convenciones como:
 - Uso de _ al inicio del nombre para indicar que es una variable interna
 - Uso de __ al inicio provoca “cierta” ocultación de la variable al modificar su nombre interno anteponiendo el de la clase
- Volveremos sobre ello cuando hablemos de las **propiedades**

Añadiendo métodos (V). Más métodos

- Continuemos añadiendo métodos a la clase Coordenada. En este caso, uno que nos devuelva la distancia entre dos coordenadas:

Coordenada
-x -y
get_x() set_x() get_y() set_y() distance()

```
import math
class Coordenada:
    . . .
    def distance(self, coord):
        x_diff = self.x - coord.x
        y_diff = self.y - coord.y
        return math.sqrt(x_diff**2 + y_diff**2)
```

otra instancia de la clase Coordenada

- El método lo podremos invocar desde un objeto o desde la propia clase:

```
p1 = Coordenada(3, 4)
p2 = Coordenada(-3, -4)
print(p1.distance(p2))
```

```
p1 = Coordenada(3, 4)
p2 = Coordenada(-3, -4)
print(Coordenada.distance(p1, p2))
```

Añadiendo métodos (VI). Métodos especiales

- ¿Qué pasaría si, del mismo modo que hacemos con las listas, los strings o los diccionarios, quisieramos imprimir nuestro objetos?

```
p1 = Coordenada(3, 4)
print(p1)
<__main__.Coordenada object at 0x7f659b14bfd0>
```

- Python nos informa de que p1 es un objeto de la clase Coordenada y nos devuelve una referencia interna a su posición en memoria. Seguramente no la respuesta que esperábamos...
- Como sabemos, nuestra clase deriva de **object**, de la que hereda varios métodos especiales como, por ejemplo, el constructor **__init__**. Otro de ellos es **__str__**, que se invoca automáticamente cuando llamamos a la función `print()`. Igual que con el constructor, podemos **sobreescribirlo** para adaptarlo a nuestras necesidades. Tiene que devolver un **string**

Añadiendo métodos (VII). Métodos especiales

- Vamos a incluir el método `__str__` para que nos devuelva un **string** con el siguiente formato: “< coord_x, coord_y >”

```
class Coordenada:  
    . . .  
    def __str__(self):  
        return '<{}, {}>'.format(self.x, self.y)
```

- Ahora el resultado sería:

```
p1 = Coordenada(3, 4)  
print(p1)  
<3, 4>
```

Añadiendo métodos (VIII). Métodos especiales

- ¿Y si quisiéramos comparar dos coordenadas?

```
p1 = Coordenada(3, 4)
p2 = Coordenada(3, 4)
print(p1 == p2)
```

False

Son objetos diferentes por lo que la comparación, por defecto, nos devolverá un valor Falso

- El mismo concepto visto anteriormente, se aplica con otros **operadores** (+,-,==,<,>,<=,>=,...) y sus **métodos especiales** asociados. Algunos son:

operador	método
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
==	<code>__eq__(self, other)</code>
<	<code>__eq__(self, other)</code>
len()	<code>__len__(self)</code>

¿Cómo modificaríamos la clase para que comparara coordenadas?

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

Añadiendo métodos (IX). Métodos especiales

- Vamos a incluir el método `__eq__` para comparar coordenadas. El valor de retorno será un **boolean**

```
class Coordenada:  
    . . .  
    def __eq__(self, other):  
        return self.x == other.x and self.y == other.y
```

- Ahora el resultado sería:

```
p1 = Coordenada(3, 4)  
p2 = Coordenada(3, 4)  
print(p1 == p2)  
True
```

Atributos de Clase (I)

- Un atributo de clase es un atributo **común** para todas las instancias de dicha clase. Dicho atributo podrá ser **accedido** y **modificado** desde cualquier objeto de la clase, así como desde la propia clase (en cierto modo, similar a los miembros *static* de Java o C++)

Fichero: u03_02.py

```
class MyClass:
    class_var = 1
    def __init__(self, val):
        self.inst_var = val
```

```
>>> from u03_02 import MyClass
>>> obj1 = MyClass(3)
>>> obj2 = MyClass(4)
>>> obj1.class_var, obj1.inst_var
(1, 3)
>>> obj2.class_var, obj2.inst_var
(1, 4)
>>> MyClass.class_var
1
>>> obj1.class_var = 99
>>> obj2.class_var
99
>>> MyClass.class_var
99
```

Atributos de Clase (y II)

- Como regla general, no deberíamos usar atributos de clase salvo en contados casos:
 - Definición de constantes o valores por defecto

```
class Circle:
    _pi = 3.14159
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return Circle._pi * self.radius**2
```

- Gestionar cierta información común a todos los objetos

```
class Person:
    _all_people = []
    def __init__(self, name):
        self.name = name
        Person._all_people.append(self)
    def remove(self):
        Person._all_people.remove(self)
```

Atributos de tipos mutables (I)

- Tenemos que tener especial cuidado cuando nuestros atributos son de tipos mutables. Recordemos que tanto en asignaciones, como en argumentos o retornos de funciones, se emplean **referencias** a dichos objetos, no los valores concretos a los que “apuntan”.
- Esto puede hacer que nos encontremos con problemas inesperados al modificar inadvertidamente un **atributo mutable** de un objeto, cuya referencia fue propagada al exterior, desde fuera de dicho objeto
- La clase de ejemplo que se muestra a continuación, tiene un atributo de tipo **lista** y, por tanto, mutable. Fíjate como en el **getter** correspondiente se devuelve la propia referencia a dicha lista, y no una referencia a una copia de la lista (como debería ser). Esto va a provocar que cambios “externos” en dicha lista afecten al atributo del objeto

Atributos de tipos mutables (II)

Fichero: u03_02.py

```
class Temp:
    def __init__(self, temp):
        self.temps = []
        self.set_temp(temp)
    def set_temp(self, temp):
        self.temp = temp
        self.temps.append(temp)
    def get_temp(self):
        return self.temp
    def get_temps(self):
        return self.temps
```

Obtenemos una copia de la lista de t^a

En realidad, la clase **no** nos devuelve una **copia**, devuelve la **misma lista** (fíjate que tiene el **mismo id**)


Si hacemos cambios en la "copia" (que no es tal), nos afecta al propio objeto

```
>>> from u03_02 import Temp
>>> t1 = Temp(21)
>>> t1.set_temp(23)
>>> print("Ta actual:", t1.get_temp())
Ta actual: 23
>>> print("Ta registradas:", t1.get_temps())
Ta registradas: [21, 23]
>>> lista_t = t1.get_temps()
>>> id(t1.temps), id(lista_t)
(139869576302280, 139869576302280)
>>> lista_t.extend([18, -2, 35])
>>> print("Ta registradas:", t1.get_temps())
Ta registradas: [21, 23, 18, -2, 35]
```

Atributos de tipos mutables (y III)

- Debemos garantizar que, cuando se trata de atributos mutables, nuestro *getter* devuelva una **copia** del mismo
- Aplicaremos lo mismo que ya vimos respecto a las copias de variables de tipos estructurados (*swallow-copy* y *deep-copy*)
- Nuestro caso lo podríamos resolver fácilmente de la forma siguiente:

```
class Temp:
    def __init__(self, temp):
        self.temps = []
        self.set_temp(temp)
    def set_temp(self, temp):
        self.temp = temp
        self.temps.append(temp)
    def get_temp(self):
        return self.temp
    def get_temps(self):
        return self.temps[:]
```



```
>>> from u03_02 import Temp
>>> t1 = Temp(21)
>>> t1.set_temp(23)
>>> print("Tª actual:", t1.get_temp())
Tª actual: 23
>>> print("Tª registradas:", t1.get_temps())
Tª registradas: [21, 23]
>>> lista_t = t1.get_temps()
>>> id(t1.temps), id(lista_t)
(139863355187016, 139863355293896)
>>> lista_t.extend([18, -2, 35])
>>> print("Tª registradas:", t1.get_temps())
Tª registradas: [21, 23]
```


Copia de objetos (I)

- Similares problemas se nos plantean cuando queremos hacer la **copia** de un objeto. Podemos abordar la copia de objetos de diversas maneras:
 - Crear nuestro propio **método** de copia:

```
class Temp:
    def __init__(self, temp):
        self.temps = []
        self.set_temp(temp)
    def set_temp(self, temp):
        self.temp = temp
        self.temps.append(temp)
    def get_temp(self):
        return self.temp
    def get_temps(self):
        return self.temps[:]

    def copy(self):
        new_temp = Temp(self.temp)
        new_temp.temps = self.temps[:]
        return new_temp
```

```
>>> from u03_02 import Temp
>>> t1 = Temp(21)
>>> t1.set_temp(23)
>>> print("Tª registradas:", t1.get_temps())
Tª registradas: [21, 23]
>>> t2 = t1.copy()
>>> t1
<U03_02.u03_02.Temp object at 0x7f79dfee4c88>
>>> t2
<U03_02.u03_02.Temp object at 0x7f79dfee4e10>
>>> id(t1.temps), id(t2.temps)
(140161424711496, 140161424710152)
>>> t2.set_temp(30)
>>> print("Tª registradas T2:", t2.get_temps())
Tª registradas: [21, 23, 30]
>>> print("Tª registradas T1:", t1.get_temps())
Tª registradas: [21, 23]
```

Copia de objetos (y II)

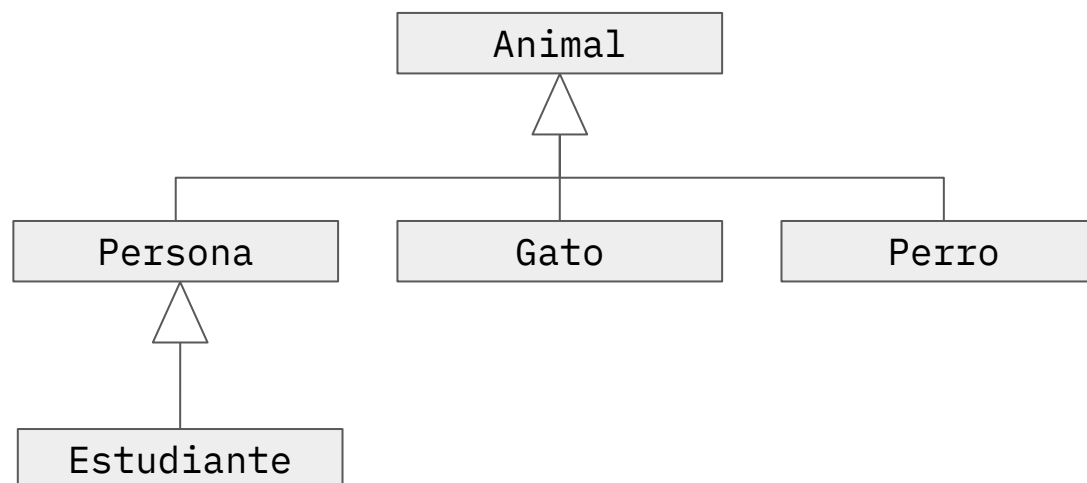
- Emplear las funciones *copy()* y *deepcopy()* del módulo *copy*, para realizar una *swallow-copy* ó *deep-copy* de nuestro objeto. El emplear una función u otra dependerá de si nuestros objetos tienen atributos mutables susceptibles verse modificados a través del **objeto copia**.
 - Tenemos la posibilidad sobrescribir (*overriding*) los métodos *__copy__* y *__deepcopy__* si queremos implementar nuestras propias versiones de dichos procesos de copia.
 - <https://docs.python.org/3/library/copy.html>

```
>>> import copy
>>> t1 = Temp(21)
>>> t2 = copy.copy(t1)
>>> t1
<U03_02.u03_02.Temp object at 0x7f79dfec4c88>
>>> t2
<U03_02.u03_02.Temp object at 0x7f79df838080>
>>> id(t1.temps), id(t2.temps)
(140161424711496, 140161424711496)
```

```
>>> import copy
>>> t1 = Temp(21)
>>> t2 = copy.deepcopy(t1)
>>> t1
<U03_02.u03_02.Temp object at 0x7f79dfec4c88>
>>> t2
<U03_02.u03_02.Temp object at 0x7f79df848438>
>>> id(t1.temps), id(t2.temps)
(140161424711496, 140161366461576)
```

Herencia (I)

- La herencia nos permite crear **jerarquías** de clases
- Estas jerarquías nos permiten agrupar en las clases padre (**superclase**) características y comportamientos comunes de sus hijos (**subclase**), al tiempo que vamos refinando los comportamientos (**especialización**) a medida que descendemos por los diferentes niveles de la jerarquía.
- Vamos a ver cómo maneja la herencia en Python. Para ello, implementaremos las clases de la siguiente jerarquía:



Herencia (II)

```
class Animal:
```

```
    def __init__(self, edad):  
        self.edad = edad  
        self.nombre = None
```

```
    def get_edad(self):  
        return self.edad
```

```
    def get_nombre(self):  
        return self.nombre
```

```
    def set_edad(self, edad):  
        self.edad = edad
```

```
    def set_nombre(self, nombre = ""):  
        self.nombre = nombre
```

```
    def __str__(self):  
        return "animal: {}: {}".format(self.nombre, self.edad)
```

getters

setters

`__init__` y `__str__`
se heredan de *object*

```
class Gato(Animal):
```

```
    def habla(self):  
        print("miau!!")
```

```
    def __str__(self):  
        return "gato: {}: {}".format(self.nombre, self.edad)
```

Hereda de *Animal*

Usa su método `__init__` y sobrescribe el método `__str__`

Añade nueva funcionalidad a través de un nuevo método

Herencia (III)

```
>>> import u03_02
>>> mi_animal = u03_02.Animal(3)
>>> print(mi_animal)
animal:None:3
>>> mi_animal.set_nombre("dude")
>>> print(mi_animal)
animal:dude:3
>>> mi_animal.get_edad()
3
>>> yin = u03_02.Cat(1)
>>> yin.habla()
miau!!
>>> yin.set_nombre("YinYang")
>>> yin.get_nombre()
'YinYang'
>>> print(yin)
gato:YinYang:1
>>> print(Animal.__str__(yin))
animal:YinYang:1
>>> blob = u03_02.Animal(1)
>>> blob.set_nombre()
>>> print(blob)
animal::1
```

Importamos el módulo donde están definidas las clases

Se busca el método en la clase (Gato). Como no se encuentra, se sube por la jerarquía hasta encontrarlo (Animal)

Podemos invocar el método *sobrescrito* del padre

Se usa el valor *por defecto* del método

Herencia (IV)

```
class Perro(Animal):  
    def habla(self):  
        print("guau!!")  
    def __str__(self):  
        return "perro: {}: {}".format(self.nombre, self.edad)
```

```
class Persona(Animal):  
    def __init__(self, nombre, edad):  
        Animal.__init__(self, edad)  
        self.set_nombre(nombre)  
        self.amigos = []  
    def get_amigos(self):  
        return self.amigos[:]  
    def add_amigo(self, amigo):  
        if isinstance(amigo, Persona) and amigo not in self.amigos:  
            self.amigos.append(amigo)  
    def habla(self):  
        print("hola!")  
    def set_nombre(self, nombre = ""):  
        self.nombre = nombre  
    def __str__(self):  
        return "persona: {}: {}".format(self.nombre, self.edad)
```

Llamada al *constructor* de la *superclase*

Nuevo atributo

Añadimos nuevos objetos sólo de tipo *Persona* (incluye subclases)

Herencia (V)

```
import random

class Estudiante(Persona):
    def __init__(self, nombre, edad, estudia=None):
        Persona.__init__(self, nombre, edad)
        self.estudia = estudia
    def get_estudia(self):
        return self.estudia
    def set_cursa(self, estudia):
        self.estudia = estudia
    def habla(self):
        r = random.random()
        if r < 0.25:
            print("tengo deberes")
        elif 0.25 <= r and r < 0.5:
            print("necesito dormir")
        elif 0.5 <= r and r < 0.75:
            print("tengo hambre!")
        else:
            print("estoy viendo la tele")
    def __str__(self):
        return "estudiante:{}: {}: {}".format(self.nombre, self.edad, self.estudia)
```

Llamada al *constructor* de la *superclase*

Nuevo atributo

Genera un *float* entre *[0, 1)*

Herencia (y VI)

```
>>> import u03_02
>>> eric = u03_02.Persona("eric", 41)
>>> joe = u03_02.Persona("joe", 32)
>>> print(joe)
persona:joe:32
>>> josh = u03_02.Estudiante("josh", 19, "Biología")
>>> fred = u03_02.Estudiante("fred", 18)
>>> print(josh)
estudiante:josh:19:Biología
>>> print(fred)
estudiante:fred:18:
>>> josh.habla()
'tengo hambre!'
>>> eric.add_amigo(joe)
>>> eric.add_amigo(josh)
>>> eric.add_amigo(fred)
>>> for amigo in eric.get_amigos():
...     print(amigo.get_nombre() + "> ", end="")
...     amigo.habla()
joe> hola!
josh> necesito dormir
fred> estoy viendo la tele
```

Enlazado *dinámico* del método (*polimorfismo*)