

# INICIACIÓN PYTHON

Julio de 2022

Germán Alonso Lascurain

[germanalonso@opendeusto.es](mailto:germanalonso@opendeusto.es)

[galonso@datavaluemanagement.es](mailto:galonso@datavaluemanagement.es)



Data Value  
management

# 0. Índice de contenidos

## 1. Introducción

- Instalación Python y componentes principales de Spyder (IDE).
- Paquetes incluidos en Anaconda
- Consola/Terminal
- Elementos de un programa Python
- Hola Mundo, primer programa.
- Tipos de variables.

# 0. Índice de contenidos

## 2. Operadores básicos

- Aritméticos
- De comparación
- De asignación
- Lógicos
- De pertenencia
- De identidad
- Comandos y operadores básicos

## 0. Índice de contenidos

3. Operando con...
  - Cadenas de texto
  - Listas
  - Tuplas
  - Dicionarios
4. Print
5. Input
6. range()
7. Operadores comunes en cadenas, tuplas, rangos y listas

## 0. Índice de contenidos

### 8. Mutación, alias y clonación de listas

- Alias
- Mutación
- Clonación

### 9. Control de flujo – Condicionales

### 10. Iteraciones

- while
- for
- for anidado
- break
- continue
- Resumen
- Ejemplo
- “Guess and check”

## 0. Índice de contenidos

### 11. Funciones

- Ámbito de las variables
- return vs. print
- Funciones como argumento
- Funciones como objetos
- Valores por defecto
- Documentación de las funciones
- Funciones anónimas

### 12. Recursión

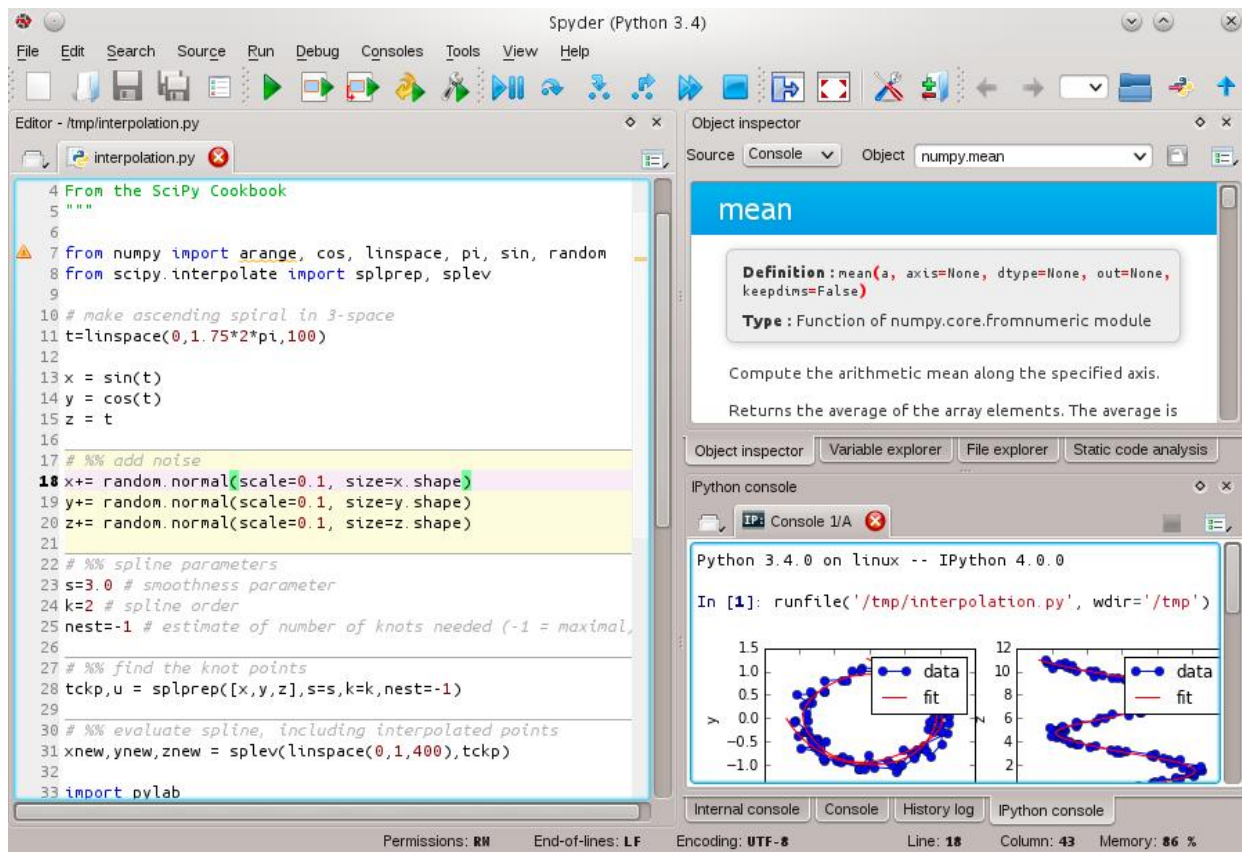
- Recursión vs. Iteración

### 13. Importar módulos

# 1. Instalación Python y componentes principales de Spyder (IDE).

- IDE = Integrated Development Enviroment = Entorno de desarrollo integrado
- Un IDE es un programa que nos ayuda a crear programas.
- 
- Un IDE consta principalmente de:
  - Editor de código.
  - Compilador o traductor de código.
  - Un depurador para encontrar errores de programación.
  - Constructor de interfaz gráfica
- Algunos ejemplos: Spyder2, Rstudio, eclipse, android studio, ...

# 1. Instalación Python y componentes principales de Spyder (IDE).



The screenshot displays the Spyder Python IDE interface (Python 3.4). The main editor window shows a script named `interpolation.py` with the following code:

```
4 From the SciPy Cookbook
5 """
6
7 from numpy import arange, cos, linspace, pi, sin, random
8 from scipy.interpolate import splprep, splev
9
10 # make ascending spiral in 3-space
11 t=linspace(0,1.75*2*pi,100)
12
13 x = sin(t)
14 y = cos(t)
15 z = t
16
17 # %% add noise
18 x+= random.normal(scale=0.1, size=x.shape)
19 y+= random.normal(scale=0.1, size=y.shape)
20 z+= random.normal(scale=0.1, size=z.shape)
21
22 # %% spline parameters
23 s=3.0 # smoothness parameter
24 k=2 # spline order
25 nest=-1 # estimate of number of knots needed (-1 = maximal,
26
27 # %% find the knot points
28 tckp,u = splprep([x,y,z],s=s,k=k,nest=-1)
29
30 # %% evaluate spline, including interpolated points
31 xnew,ynew,znew = splev(linspace(0,1,400),tckp)
32
33 import pylab
```

The Object inspector panel on the right shows the `mean` function from `numpy.core.fromnumeric`. The definition is: `mean(a, axis=None, dtype=None, out=None, keepdims=False)`. The description states: "Compute the arithmetic mean along the specified axis. Returns the average of the array elements. The average is

The IPython console at the bottom shows the command `runfile('/tmp/interpolation.py', wdir='/tmp')` and displays two plots. The left plot shows a 3D scatter plot of data points (blue dots) and a fitted spline (red line). The right plot shows a 2D scatter plot of data points (blue dots) and a fitted spline (red line).

At the bottom of the IDE, the status bar shows: Permissions: **RM**, End-of-lines: **LF**, Encoding: **UTF-8**, Line: **18**, Column: **43**, Memory: **86 %**.





## 1. Instalación Python y componentes principales de Spyder (IDE).

¿Qué es un IDE?

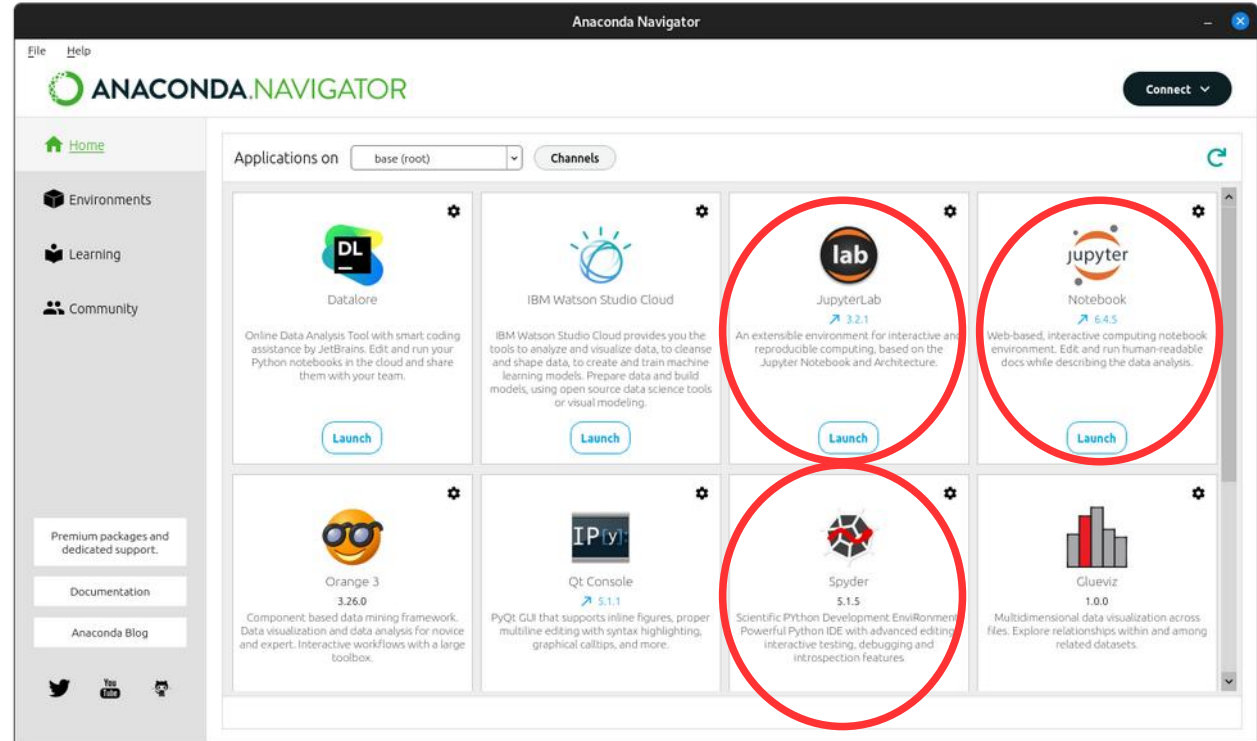


# 1. Instalación Python y componentes principales de Spyder (IDE).

- Podemos instalar Python “a mano” o bien mediante la ayuda de una de las siguientes distribuciones (en Windows):
  - 1) Anaconda
  - 2) WinPython
  - 3) Python (x,y)
- Las principales distribuciones Linux ya incorporan Python por defecto, por lo que no es necesario instalarlo.
- Nosotros instalaremos Python a través de Anaconda:  
**`https://www.anaconda.com/download`**

# 1. Instalación Python y componentes principales de Spyder (IDE).

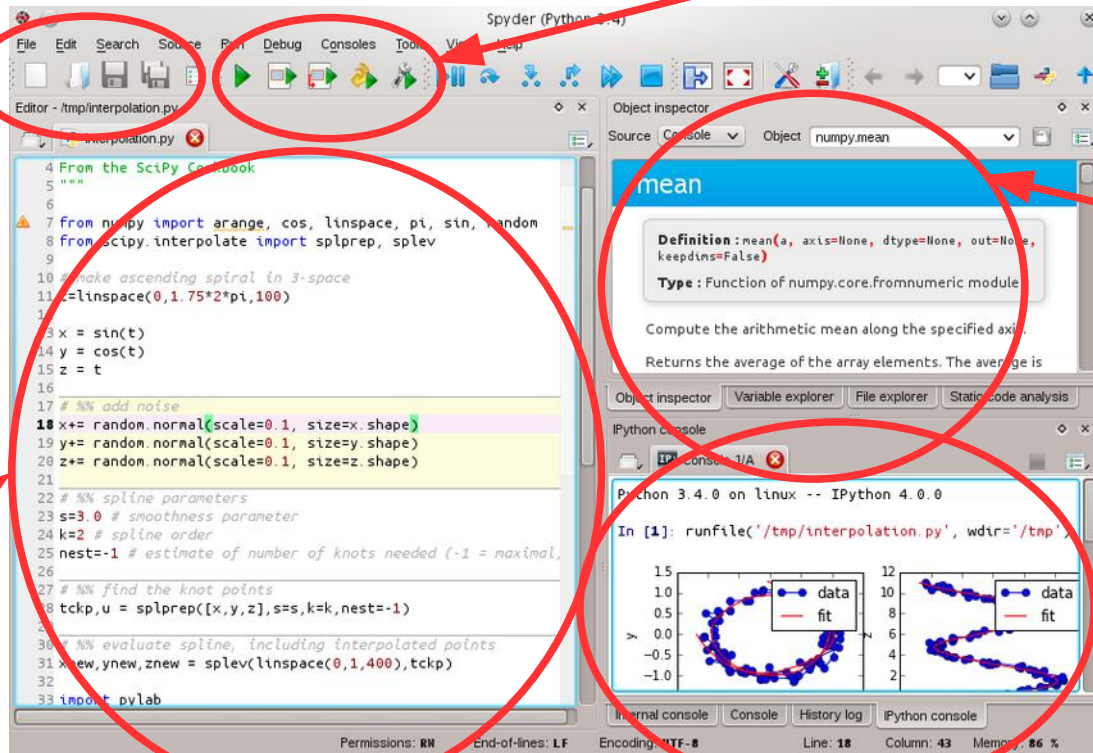
- Anaconda nos permite instalar diferentes elementos a través de Anaconda Navigator



# 1. Instalación Python y componentes principales de Spyder (IDE).

Nuevo, Abrir, Guardar

Ejecución de programas



Inspector de objetos

Terminal de Python

Editor de programas

# 1. Instalación Python y componentes principales de Spyder (IDE).

- Podemos acceder a la ayuda y a un tutorial de Spyder (en inglés), a través de:
  - Ayuda – Documentación Spyder (F1)
  - Ayuda – Tutorial de Spyder.

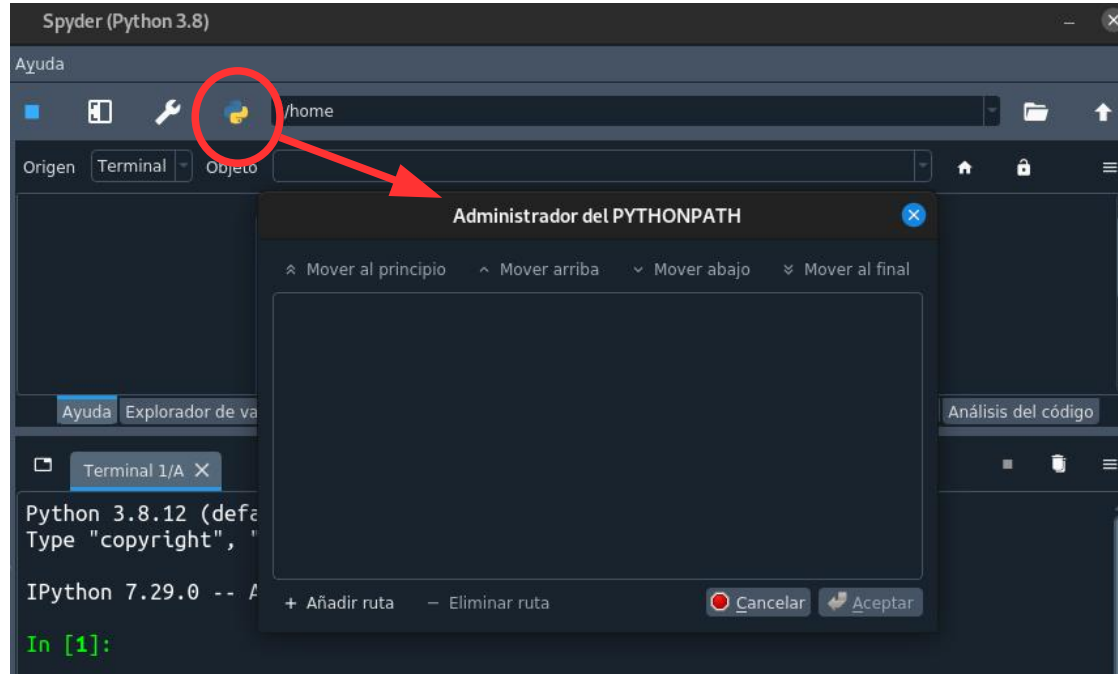
- Un opción muy interesante es activar la ayuda en el editor y en el terminal.

A través del mismo, el IDE nos permite autocompletar cada comando/función de Python a medida que la tecleamos.



# 1. Instalación Python y componentes principales de Spyder (IDE).

- Ruta por defecto de nuestros archivos. Permite configurar Python con las rutas donde ubicamos nuestros ficheros.
- Herramientas – Administrador del PYTHONPATH



## 1. Paquetes incluidos en Anaconda

Paquete	Web	Descripción
NumPy	<a href="https://numpy.org">numpy.org</a>	Añade soporte sobre vectores y matrices.
SciPy	<a href="https://scipy.org">scipy.org</a>	Contiene módulos de álgebra lineal, interpolación, optimización, ...
Matplotlib	<a href="https://matplotlib.org">matplotlib.org</a>	Biblioteca gráfica 2D, 3D.
Pandas	<a href="https://pandas.pydata.org">pandas.pydata.org</a>	Proporciona estructuras de datos similares a los dataframes en R. Depende de Numpy.
Seaborn	<a href="https://seaborn.pydata.org">seaborn.pydata.org</a>	Librería de visualización de datos, basada en matplotlib.
Bokeh	<a href="https://bokeh.pydata.org">bokeh.pydata.org</a>	Framework de visualización de grandes volúmenes de datos de manera interactiva en navegadores web.
Scikit-Learn	<a href="https://scikit-learn.org/stable">scikit-learn.org/stable</a>	Librería para tareas de Machine Learning, que incluye entre otros clustering, random forest, k-means, regresión, ...

## 1. Paquetes incluidos en Anaconda

Paquete	Web	Descripción
NLTK	<a href="http://nltk.org">nltk.org</a>	Librería para el procesamiento natural de lenguaje.
Jupyter Notebook	<a href="http://jupyter.org">jupyter.org</a>	Aplicación web que permite crear y compartir documentos que contienen código, visualizaciones, comentarios, ...
R essentials	<a href="http://conda.pydata.org/docs/r-with-conda.html">conda.pydata.org/docs/r-with-conda.html</a>	Permite el uso de una 80 librerías de R en Python

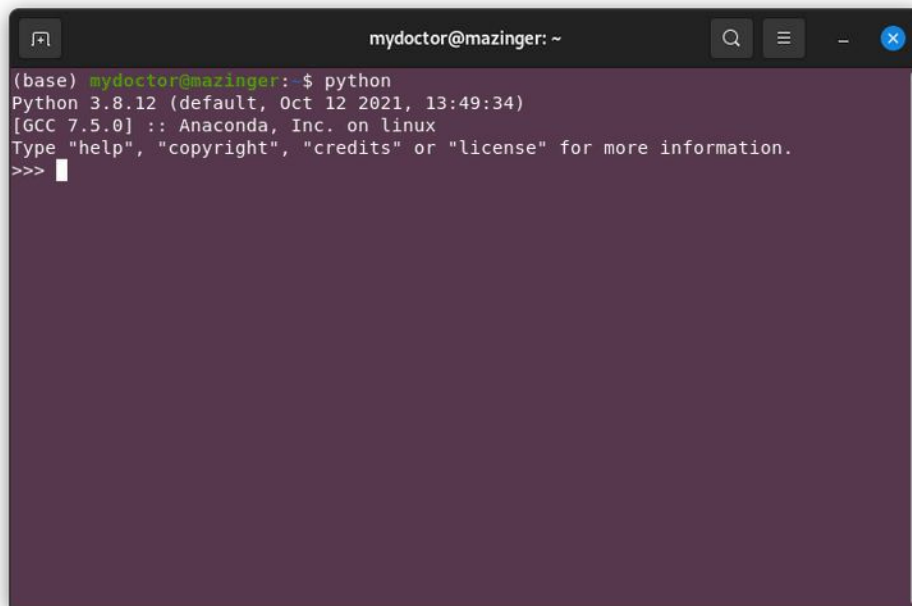


# 1. Instalación Python y componentes principales de Spyder (IDE).

- Listar los paquetes instalados en anaconda, junto con su versión:
  - 1) Abrir un terminal Anaconda (Inicio – AnacondaX (x-bit) - Anaconda Prompt )
  - 2) Escribir: **conda list**

# 1. Consola / Terminal

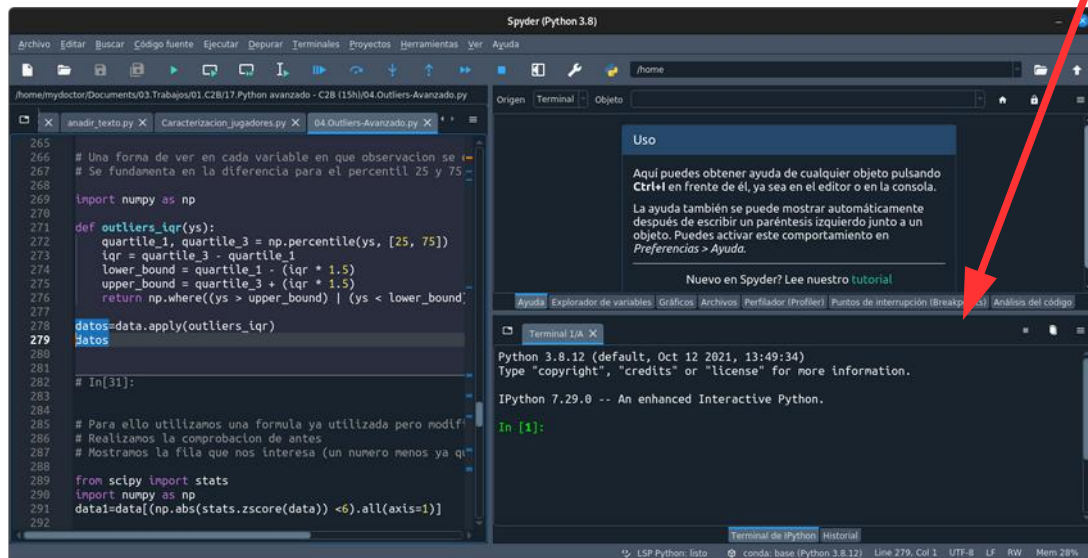
- Con Python instalado a través de Anaconda, podemos hacer uso de la consola de 2 maneras:
  - Abrir un Terminal de Anaconda, escribir python y pulsar Enter.



```
mydoctor@mazing: ~  
(base) mydoctor@mazing:~$ python  
Python 3.8.12 (default, Oct 12 2021, 13:49:34)  
[GCC 7.5.0] :: Anaconda, Inc. on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> 
```

# 1. Consola / Terminal

- Con Python instalado a través de Anaconda, podemos hacer uso de la consola de 2 maneras:
  - Abrir Spyder. La consola está situada en la esquina inferior derecha del IDE.



## 1. Consola / Terminal

- A través de la consola podemos ejecutar scripts Python directamente o ejecutar comandos.
- Abrir una consola (a través del IDE o a través de Anaconda Prompt en Windows) y escribir:

`print ("Hola Mundo")` → Pulsar **Enter** (A través IDE)

`python \ruta\al\script\holamundo.py` → Pulsar **Enter** (A través de Anaconda Prompt)

- El uso del terminal del IDE es más conveniente, ya que el IDE incluye entre otros un depurador de código que de forma gráfica nos muestra los errores de sintaxis cometidos.
- Para abandonar el terminal python abierto desde el terminal de Windows, teclear `quit()`.

# 1. Elementos de un programa Python

- Un programa de Python es un fichero de texto (normalmente guardado con el juego de caracteres UTF-8) que contiene expresiones y sentencias del lenguaje Python.
- Esas expresiones y sentencias se consiguen combinando los elementos básicos del lenguaje.
- Este tipo de ficheros se conocen como Scripts.
- Para ejecutar un script, debemos invocarlo mediante un intérprete y por tanto no es necesario compilarlo. El script se ejecuta hasta su finalización de manera descendente.

# 1. Elementos de un programa Python

- El lenguaje Python está formado por elementos de diferentes tipos:
- Palabras reservadas (keywords)

<b>and</b>	<b>exec</b>	<b>not</b>
<b>assert</b>	<b>finally</b>	<b>or</b>
<b>break</b>	<b>for</b>	<b>pass</b>
<b>class</b>	<b>from</b>	<b>print</b>
<b>continue</b>	<b>global</b>	<b>raise</b>
<b>del</b>	<b>if</b>	<b>return</b>
<b>del</b>	<b>import</b>	<b>try</b>
<b>elif</b>	<b>in</b>	<b>while</b>
<b>else</b>	<b>is</b>	<b>with</b>
<b>except</b>	<b>lambda</b>	<b>yield</b>

# 1. Elementos de un programa Python

- Funciones integradas (built-in functions) → `abs()`, `bool()`, `chr()`, `len()`, `min()`, ...
- Literales → números y cadenas de texto
- Operadores → `+`, `-`, `*`, `**`, `/`, `//`, `%`, ...
- Delimitadores → `'`, `"`, `#`, `\`, `(`, `)`, ...
- Identificadores → Son palabras utilizadas para nombrar elementos creados por el usuario

# 1. Hola Mundo, primer programa

- 2 maneras para ejecutar un programa Python.
  - ◆ Desde la consola del ordenador
  - ◆ Desde la consola del IDE

Ej:

- En el editor de Spyder, escribir:

```
print ("Hola Mundo")
```

Pulsar sobre



y comprobar el resultado en la consola de Spyder.

- Guardar el programa generado en el Escritorio como **holamundo.py**  
Abrir un Terminal en Windows, desplazarse a la carpeta Escritorio  
Escribir **python3 holamundo.py** y comprobar el resultado.



# 1. Tipos de variables

- A la hora de trabajar con variables, en función del tipo de dato a contener, podemos encontrarnos con los siguientes tipos:
  1. **Números.**
  2. **Cadenas de texto.** Se encierran entre comillas simples o dobles.
  3. **Listas de datos.** Pueden contener números y/o caracteres y se encierran entre corchetes.
  4. **Tuplas.** Igual que una lista, es una secuencia ordenada de elementos de diferente tipo. Se encierran entre paréntesis.
  5. **Diccionarios.** Se encierran entre llaves.

# 1. Tipos de variables

- Cadenas de texto:

- Una cadena de texto es un conjunto de caracteres representados entre comillas (dobles o sencillas). Las cadenas de texto, son sensibles a las mayúsculas.
- Son INMUTABLES. No podemos cambiar individualmente un elemento de la cadena. Debemos hacerlo en conjunto.
- Son iterables.

*Ej: "Hola Mundo"*

*'abcde'*

*"En un lugar de la Mancha, de cuyo nombre no quiero acordarme, no ha mucho tiempo que vivía un hidalgo de los de lanza en astillero, adarga antigua, rocín flaco y galgo corredor"*

*"a1b2c3d4e5?+&"*

# 1. Tipos de variables

- Listas:

- Lista de cadenas de texto

```
["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado"]
```

- Lista de números

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

- Lista mixta

```
["Lunes", 1, "Mayo", 2018]
```

- Lista que contiene listas

```
[["Alberto", 1.75, 80], ["Ana", 1.70, 65], ["María", 1.90, 80]]
```

- Listas anidadas

```
[["Stanley Kubrick", ["Senderos de Gloria", 1957]], ["Woody Allen", ["Hannah y sus hermanas", 1986]]]
```

# 1. Tipos de variables

- Listas:

- Los datos dentro de una lista, van ordenados empezando desde el 0.  
Supongamos una lista con 9 elementos, el número 47 ocupa la posición 6 dentro de la lista.

`[1,22,3,14,5,76,47,28,19]`

- Los elementos de una lista pueden referenciarse mediante este índice.

Ej: `a = [1,22,3,14,5,76,47,28,19]`

`a [1] → nos devuelve 22`

`a [8] → nos devuelve 19`

`a [-1] → nos devuelve 19`

# 1. Tipos de variables

- Tuplas:

- Son similares a las listas, pero son **INMUTABLES**.
- Son iterables.
- Se definen igual que una lista, pero encerrando el conjunto de datos entre ( ).
- No pueden añadirse ni eliminarse elementos a una tupla.
- No pueden buscarse elementos en una tupla, pero sí podemos comprobar si un elemento está en la tupla a través de *in*.

# 1. Tipos de variables

- **Tuplas:**

¿Cuándo usar tuplas?

- Necesitemos velocidad para acceder a los datos, ya que son más rápidas de las listas.
- Para un conjunto constante de valores, que solo hemos de recorrer, es más eficiente el uso de tuplas.
- Para formatear cadenas.

*Normalmente las tuplas contienen más de un elemento,*

```
x = (1, 2, "tres", "cuatro", 5)
```

*pero si solo tuviera un elemento debemos incluir una coma al final*

```
x = (1,) → si no lo hacemos así, Python considerará x como un entero
```

```
x = ("dos",) → si no lo hacemos así, Python considerará x como una cadena de texto
```

# 1. Tipos de variables

- Dictionarios:

- Al igual que en un diccionario convencional, un diccionario Python es una palabra que tiene asociado algo.
- Al contrario que en las listas, los diccionarios **NO TIENEN ORDEN**.
- Se definen encerrando sus elementos entre { }
- Los diccionarios son **MUTABLES**.
- Los diccionarios son **iterables** (pueden recorrerse).

# 1. Tipos de variables

- Diccionarios:

Ej:     { 'Cómico1' : "Chico Marx",     'Cómico2' : 'Harpo Marx',  
          'Cómico3' : 'Groucho Marx' }



- Los diccionarios constan de claves (keys) y definiciones (values).
- En un mismo diccionario, **no puede haber 2 claves iguales.**



# 1. Tipos de variables

- Dicionarios:

- Valores:

- ✓ Los valores, pueden ser de cualquier tipo, inmutables o mutables.
    - ✓ Puede haber valores duplicados.
    - ✓ Pueden ser listas, o incluso otros diccionarios.

- Claves:

- ✓ Son únicas dentro del diccionario.
    - ✓ Son inmutables
    - ✓ Tipos soportados: `int`, `float`, `string`, `tuple`, `bool` (son inmutables)
    - ✓ Cuidado al usar números de coma flotante como índice.

- ✓ Los elementos de un diccionario no tienen orden.

## 1. Tipos de variables

LISTAS

0	Elemento1
1	Elemento2
2	Elemento3
3	Elemento4
...	...

DICCIONARIOS

KEY1	Valor1
KEY2	Valor2
KEY3	Valor3
KEY4	Valor4
...	...

# 1. Tipos de variables

- Dictionarios:

- La idea subyacente a un diccionario, es poder localizar rápidamente información, sin necesidad de conocer la posición en el índice de dicha información.

*Es más fácil crear un índice de color de pelo (castaño, rubio, pelirrojo, ...) para poder localizar a las personas pelirrojas que tener que recordar, si los pelirrojos estaban en el índice 1, 80 ó 2013.*

*De esta manera podemos trabajar con diccionarios (colecciones de listas, pej) con un índice que sea representativo para nuestro trabajo con esos datos.*

# 1. Tipos de variables

- Dicionarios vs. listas

Diccionarios	Listas
Colección emparejada de claves y valores.	Secuencia ordenada de elementos.
Búsqueda de elementos a través de otros.	Búsqueda de elementos a través de un índice.
Datos no ordenados.	Datos ordenados según un índice.
La clave puede ser cualquier tipo inmutable.	El índice es un entero.

# 1. Tipos de variables

- Conversión de datos a diferentes formatos:

<code>int(x)</code>	Convierte x a entero
<code>float(x)</code>	Convierte x a coma flotante
<code>complex(real)</code>	Convierte x a complejo
<code>str(x)</code>	Convierte x a cadena de texto
<code>tuple(s)</code>	Convierte x a tupla
<code>list(s)</code>	Convierte x a lista
<code>dict(d)</code>	Convierte x a diccionario
<code>chr(x)</code>	Convierte un entero a carácter ASCII
<code>ord(x)</code>	Convierte un carácter ASCII a entero
<code>hex(x)</code>	Convierte un entero a cadena hexadecimal
...	

# 1. Tipos de variables

- Conversión de datos a diferentes formatos:

Ejemplos:

```
a = "2"
```

```
type (a)
```

```
a = int (a)
```

```
type (a)
```

```
a = "2.0"
```

```
type (a)
```

```
a = float (a)
```

```
type (a)
```

# 1. Tipos de variables

- Conversión de datos a diferentes formatos:

Ejemplos:

```
a = 33
```

```
type (a)
```

```
a = chr (a)
```

```
type (a)
```

```
a
```

```
a = 33
```

```
type (a)
```

```
a = str(a)
```

```
type (a)
```

## 2. Operadores básicos Aritméticos

Operador	Descripción	Ejemplo
+	Suma	$a + b = 10$ $"a" + "b" = "ab"$
-	Resta	$a - b = 10$
*	Multiplicación	$a * b = 10$ $3 * "a" = "aaa"$
/	División	$a / b = 10$
%	Devuelve el resto de una división	$b \% a = 0$
**	Eleva un número a la potencia	$a ** b = a^b$
//	Devuelve la parte entera de la división	$5 // 2 = 2$



## 2. Operadores básicos de Comparación

Operador	Descripción	Ejemplo
<b>==</b>	Devuelve True, si el valor de los 2 operadores es igual	<b>a == b</b>
<b>!=</b>	Devuelve True, si el valor de los 2 operadores NO es igual	<b>a != b</b>
<b>&lt;&gt;</b>	Igual a !=	<b>a &lt;&gt; b</b>
<b>&gt;</b>	Devuelve True, si el operador izquierdo es mayor al derecho	<b>a &gt; b</b>
<b>&lt;</b>	Devuelve True, si el operador izquierdo es menor al derecho	<b>a &lt; b</b>
<b>&gt;=</b>	Devuelve True, si el operador izquierdo es mayor o igual al derecho	<b>a &gt;= b</b>
<b>&lt;=</b>	Devuelve True, si el operador izquierdo es menor o iguala al derecho	<b>a &lt;= b</b>

## 2. Operadores básicos de Asignación

Operador	Descripción	Ejemplo
=	Asigna un valor a una variable	<code>c = a + b</code>
+=	Suma a la parte izquierda del operando, la parte derecha	<code>c += a</code> ( <code>c = c + a</code> )
-=	Resta a la parte izquierda del operando, la parte derecha	<code>c -= a</code> ( <code>c = c - a</code> )
*=	Multiplica a la parte izquierda del operando, la parte derecha	<code>c *= a</code> ( <code>c = c * a</code> )
/=	Divide a la parte izquierda del operando, la parte derecha	<code>c /= a</code> ( <code>c = c / a</code> )

## 2. Operadores básicos Lógicos

Operador	Descripción	Ejemplo
<code>and</code>	Conjunción	<code>1 &gt; 2 and 3 &gt; 2 → False</code>
<code>or</code>	Disyunción	<code>1 &gt; 2 or 3 &gt; 2 → True</code>
<code>not</code>	Negación	<code>Not 3 → True</code> cuando lo que se evalúa no es igual a 3

## 2. Operadores básicos de Pertenencia

Operador	Descripción	Ejemplo
<code>in</code>	Devuelve True si encuentra una variable dentro de una secuencia de datos	<code>a in b</code>
<code>not in</code>	Devuelve True si NO encuentra una variable dentro de una secuencia de datos	<code>a not in b</code>

## 2. Operadores básicos de Identidad

Operador	Descripción	Ejemplo
<code>is</code>	Devuelve 1 si ambos operadores a comparar son iguales	<code>a is b</code>
<code>is not</code>	Devuelve 0 si ambos operadores a comparar son diferentes	<code>a is not b</code>

### 3. Comandos y operadores básicos

- Operando con cadenas de texto:

- Suma de cadenas de texto:

```
saludo = "Hola"  
sujeto = "Mundo"  
frase = saludo + " " + sujeto  
→ "Hola Mundo"
```

- Una cadena de texto, está compuesta de n-elementos. Podemos acceder individualmente (o en un intervalo) a dichos elementos.

<code>frase [0] → H</code>	<code>frase[:6] = Hola M</code>
<code>frase [5] → M</code>	<code>frase[2:7] = la Mu</code>
<code>frase [-1] → o</code>	<code>frase[2:] = la Mundo</code>

```
frase [:] → ¿?
```

### 3. Comandos y operadores básicos

- Operando con cadenas de texto:

- Particionado de cadenas de texto [inicio:fin:pasos] :

```
s = 'abcdefgh'
```

```
s[::-1] → devuelve 'hgfedcba'
```

```
s[3:6] → devuelve 'def'
```

```
s[-1] → devuelve 'h'
```

- Si bien podemos acceder al dato contenido en una posición de la cadena, NO PODEMOS MODIFICARLO individualmente.

```
s = 'abcdefgh'
```

```
s[0] → 'a'
```

- *Si intento `s[0] = 'A'`, nos dará Error, ya que las cadenas de texto son INMUTABLES. Para modificar una cadena, debemos modificarla en su conjunto.*

### 3. Comandos y operadores básicos

- Operando con cadenas de texto:
  - ¿Existe la letra 'i' en la cadena de texto?

```
cadena = "abcdefghijkl"
```

```
"i" in cadena
```



### 3. Comandos y operadores básicos

- Operando con cadenas de texto:
  - Convertir una cadena texto en una lista → `list()`

Convierte una cadena de texto, en una lista, donde cada elemento está compuesto por un carácter.

```
cadena = "Hola"  
list (cadena) → ['H', 'o', 'l', 'a']
```

### 3. Comandos y operadores básicos

- Operando con cadenas de texto:

- Dividir una cadena `.split('elemento_divisor')`

Divide una cadena de texto por el elemento divisor. El resultado es una lista.  
Si no se indica divisor, divide la cadena por los espacios.

```
cadena = "Alberto es mayor que Pedro"
```

```
cadena.split('es mayor que') → ['Alberto ', ' Pedro']
```

```
cadena.split('mayor') → ['Alberto es ', ' que Pedro']
```

```
cadena.split() → ['Alberto', 'es', 'mayor', 'que', 'Pedro']
```

- Nota: cadena NO cambia cada vez que ejecutamos `.split` y por tanto NO muta.

### 3. Comandos y operadores básicos

- Operando con cadenas de texto:

- Escribe un código que compruebe si para una cadena de texto dada (input), existen las letras “a” y “e”, de manera que devuelva:

Si solo existe la “a” → La cadena contiene solo la “a”

Si solo existe la “e” → La cadena contiene solo la “e”

Si existen ambas → La cadena contiene ambas letras

### 3. Comandos y operadores básicos

- Operando con cadenas de texto:
  - Información ampliada sobre cadenas de texto:  
<https://docs.python.org/3/tutorial/introduction.html#strings>

### 3. Comandos y operadores básicos

- Operando con listas:

```
a = [] → lista vacía
```

```
b = [1,3,"hola",4]
```

- Cada elemento de la lista, está referenciado con un índice.

```
len (b) → devuelve 4
```

```
b[0] → devuelve 1
```

```
b[1]+1 → devuelve 4
```

```
b[4] → devuelve error, número fuera del rango del  
índice
```

### 3. Comandos y operadores básicos

- Operando con listas:

Podemos usar variables para referenciar índices.

```
i = 2
```

```
b [i]    → devuelve "hola"
```

```
i = 5
```

```
b[i-4]   → devuelve 3
```

Nota: Cuando programemos, debemos asegurarnos de que las variables que usemos, trabajan en el rango esperado (en este caso el tamaño de la lista).

### 3. Comandos y operadores básicos

- Operando con listas:

Podemos modificar elemento/s de una lista (no en las tuplas)

```
b [3] = "caracola"
```

```
b → devuelve [1, 3, 'hola', 'caracola']
```

### 3. Comandos y operadores básicos

- Operando con listas:

Podemos realizar iteraciones sobre listas

Sumar de los elementos de una lista → `lista = [1,2,3,5,6]`

Método 1

```
total = 0
for i in range(len(lista)):
    total += lista[i]
print (total)
```

Nota: el primer elemento de una lista tiene índice = 0.  
range(n) varía desde 0 a n-1  
Podemos verlo con el comando  
`print(list(range(len(lista))))`

Método 2

```
total = 0
for i in lista:
    total += i
print(total)
```

Estamos iterando sobre los propios elementos de la lista directamente.



### 3. Comandos y operadores básicos

- Operando con listas:

Agregación de nuevos elementos a una lista.

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

```
lista.append(4)
```

```
lista → [1,2,3,4]
```

Objeto lista

Método o función del objeto lista. Cada objeto tiene diferentes funciones y métodos en función de su tipo. Sintaxis estandar:  
`nombre_objeto.función_o_método ()`

Nota: En este caso, tras el append, **lista ha cambiado (mutado)** y ya no es el mismo objeto.

### 3. Comandos y operadores básicos

- Operando con listas:

Agregación de nuevos elementos a una lista.

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

```
lista.extend([0,6])
```

```
lista → [1,2,3,0,6]
```

- Nota: En este caso, tras el extend, **lista ha cambiado (mutado)** y ya no es el mismo objeto.

### 3. Comandos y operadores básicos

- Operando con listas:

.append() vs. .extend()

. **append()** : **Añade** el argumento del método, como un **elemento único al final** de una lista. **Añade** a un objeto inicial, otro objeto (argumento).

. **extend()** : **Concatena** (extiende) a una lista, **otro objeto iterable** (cadena texto, lista, tupla o diccionario).

### 3. Comandos y operadores básicos

- Operando con listas:

Concatenación de listas, mediante el uso del operador +

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

```
lista2 = [4,5,6]
```

```
lista3 = lista + lista2
```

Devuelve → `lista3 = [3,2,1,4,5,6]`

Nota: lista y lista2 NO cambian (no mutan).

### 3. Comandos y operadores básicos

- Operando con listas:

Eliminar elementos de listas → `del (lista[index])`

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

```
del (lista[2]) → lista = [3,2]
```

Nota: lista ha cambiado y por tanto ha mutado.

### 3. Comandos y operadores básicos

- Operando con listas:

Eliminar elemento del final de una lista → `.pop()`

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

`lista.pop()` → Devuelve el elemento eliminado (1) y muta lista a `[3,2]`

Nota: lista ha cambiado y por tanto ha mutado.

### 3. Comandos y operadores básicos

- Operando con listas:

Eliminar elemento dentro de una lista → `.remove()`

Borra primera ocurrencia en la lista.

Si el elemento a eliminar no está en la lista, devuelve error.

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

```
lista.remove(3) → lista = [2,1,4,5,6,3]
```

```
lista.remove(5) → lista = [2,1,4,6,3]
```

```
lista.remove(8) → list.remove(x): x not in list
```

Nota: lista cambia cada vez que ejecutamos `.remove` y por tanto muta.

### 3. Comandos y operadores básicos

- Operando con listas:

Convertir una lista de caracteres en una cadena → `.join()`

Aparte de convertir en cadena el contenido de una lista, podemos también definir el separador entre dichos elementos.

```
lista = ['Alberto', 'es', 'mas', 'alto', 'que', 'Pedro']  
''.join(lista) → 'AlbertoesmasaltoquePedro'  
' '.join(lista) → 'Alberto es mas alto que Pedro'  
'_'.join(lista) → 'Alberto_es_mas_alto_que_Pedro'
```

Nota: lista no cambia cada vez que ejecutamos `.join` y por tanto no muta.



### 3. Comandos y operadores básicos

- Operando con listas:

Ordenar una lista → `.sort()` , `sorted()` y `.reverse()`

```
lista = [0,2,4,6,1,3,5,7]
```

```
sorted(lista)
```

→ Devuelve `[0, 1, 2, 3, 4, 5, 6, 7]`

→ `lista` No muta

```
lista.sort()
```

→ No devuelve nada, PERO

→ `lista` muta → `[0, 1, 2, 3, 4, 5, 6, 7]`

```
lista.reverse()
```

→ No devuelve nada, PERO

→ `lista` muta → `[7, 6, 5, 4, 3, 2, 1, 0]`

### 3. Comandos y operadores básicos

- Operando con listas:

Información ampliada sobre listas:

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

### 3. Comandos y operadores básicos

- Operando con tuplas:

Las operaciones que podemos realizar, son básicamente las mismas que para cadenas de texto, con la salvedad de que no podemos modificar (por separado) un elemento de la tupla. Hay que hacerlo en conjunto.

```
tupla = () → type(tupla)
```

```
tupla = (1, "dos", 3)
```

```
t[0] → 1
```

```
(1,"dos",3) + (4,"cinco") → (1,"dos",3,4,"cinco")
```

```
tupla [1:2] → ("dos",)
```

(La coma significa que la tupla tiene 1 elemento y no es un elemento sencillo de otro tipo de dato)

### 3. Comandos y operadores básicos

- Operando con tuplas:

```
tupla [1:3] → ("dos",3)
```

```
t[1] = 4 → error. No podemos modificar el objeto
```

- Una propiedad interesante de las tuplas es que permiten el intercambio de valores entre variables de manera rápida:

x e y, queremos intercambiar sus valores.

```
x = y
```

**y = x**, aquí habremos perdido el valor original de x. Es necesario el uso de una variable temporal.

Con las tuplas:

```
(x,y) = (y,x)
```

### 3. Comandos y operadores básicos

- Operando con tuplas:

Otra propiedad interesante, es que las tuplas forman un conjunto  $\rightarrow (x, y)$ .

Puedo definir una tupla en cualquier momento y la tupla en su conjunto será tratada por Python como un todo.

➤ Esto nos permite, por ejemplo, que una función nos devuelva más de un valor.

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

```
(cociente, resto) = cociente_resto (5,6)
```

### 3. Comandos y operadores básicos

- Operando con diccionarios:
- Comprobar si un elemento existe dentro del diccionario.  
`"P1" in diccionario`

```
diccionario → {'P2': 'Amaia',  
               'P3': 'Nekane',  
               'P4': 'Ekaitz'}
```

Devuelve **True**, si el nombre **P1** está en el diccionario. En el ejemplo devolvería **False**

### 3. Comandos y operadores básicos

- Operando con diccionarios:
- Eliminar elementos de un diccionario.

`del(diccionario['P1'])` → no devuelve nada

`diccionario` → `{'P2': 'Amaia', 'P3': 'Nekane', 'P4': 'Ekaitz'}`

NOTA: Todas las operaciones las realizamos contra las keys, no contra los values.

### 3. Comandos y operadores básicos

- Operando con diccionarios:

- Añadir elementos:

```
diccionario = {'P1':"Irati", 'P2':'Amaia', 'P3':'Nekane'}
```

- Añadir una nueva persona al diccionario,

```
diccionario ['P4'] = 'Ekaitz'
```

```
diccionario → {'P1': 'Irati',  
               'P2': 'Amaia',  
               'P3': 'Nekane',  
               'P4': 'Ekaitz'}
```



### 3. Comandos y operadores básicos

- Operando con diccionarios:

- Obtener las claves (keys) de un diccionario.

```
diccionario.keys() → dict_keys(['P2', 'P3', 'P4'])
```

- Obtener los valores (values) de un diccionario.

```
diccionario.values() → dict_values(['Amaia', 'Nekane', 'Ekaitz'])
```

- Obtener el valor de una clave.

```
diccionario.get ('P4')
```

```
diccionario ['P4']
```

### 3. Comandos y operadores básicos

- Operando con diccionarios:
- Obtener la clave correspondiente a un valor.

```
def clave (diccionario, valor):  
    for a in diccionario.keys():  
        if diccionario [a] == valor:  
            return a  
    else:  
        print("El valor " + valor + " no está en el diccionario")
```

### 3. Comandos y operadores básicos

- Operando con diccionarios:
- Ej: Crear un 'diccionario' que incluya la tabla de frecuencias de las palabras de un texto dado.  
*Nota: Para este ejemplo, texto, debería contener una sucesión de cadenas (palabras) (archivo de texto plano).*

```
quijote = open('quijote.txt', 'r')
contenido = quijote.read()
lista_palabras = contenido.split()

def tabla_frecuencias (texto):
    diccionario = {}
    for palabra in texto:
        if palabra in diccionario:
            diccionario [palabra] += 1
        else:
            diccionario [palabra] = 1
    return diccionario

resultado = tabla_frecuencias(lista_palabras)
```

## 4. Comandos y operadores básicos

- `print` (imprimir información en pantalla):

- Imprimir cadenas de texto:

```
print (saludo, persona)
```

→ `Hola Mundo`

```
print (saludo + " " + sujeto) → concatena cadenas
```

→ `Hola Mundo`

```
print (saludo + sujeto)
```

→ `HolaMundo`

### 3. Comandos y operadores básicos

- `print` (imprimir información en pantalla):
- Método `format()`  
Las llaves y caracteres dentro de los formatos de campo, son reemplazadas con los objetos pasados en el método `str.format()`

```
print ("Esta {sujeto} es {adjetivo}.".format(sujeto="comida", adjetivo="muy buena"))
```

Esta comida es muy buena.

```
for x in range(1,11):  
    print("{0:2d} {1:3d} {2:4d}".format(x, x*x, x*x*x))
```

```
1      1      1  
2      4      8  
3      9     27  
...
```

## 5. Comandos y operadores básicos

- input (Introducir información en el programa):
- A través de comando input, podemos hacer que el programa solicite que le proporcionemos determinada información para alguna variable que estemos utilizando.

```
texto = input ("Introduce un nombre")  
print ("Escribo el nombre 4 veces...")  
print (4*texto)
```

## 5. Comandos y operadores básicos

- input (Introducir información en el programa):

Input trabaja con datos en formato STRING. Por tanto cualquier dato numérico que introduzcamos, se convertirá a texto.

*Pej: Para convertir un input en dato numérico*

```
numero = int(input ("Introduce un numero"))  
print ("tu número * 4")  
numero = numero * 4  
print (numero)
```

`type(numero)` → comprobamos a través de la consola, que tipo de dato es número  
(int)

## 5. Comandos y operadores básicos

- input (Introducir información en el programa):

No es necesario incluir las dobles comillas de apertura y cierre en input. Si lo hacemos, Python considerará las comillas como parte del texto introducido.

*Ej: Crear y ejecutar código en Spyder, donde nos pida introducir una cadena de texto. Proporcionar al programa una cadena de texto encerrada entre comillas. Imprimir el resultado en pantalla.*



## 6. Comandos y operadores básicos

- `range()`:

Devuelve una lista con un rango de datos.

`range(5)` → `[0,1,2,3,4]`

`range(2,6)` → `[2,3,4,5]`

`range(5,2,-1)` → `[5,4,3]`

Muy útil cuando trabajemos con **for**:

```
for a in range (5):
```

```
    <expresiones>
```

```
print (range(5)) devuelve
```

```
range(0, 5)
```

```
print(list(range(5))) devuelve
```

```
[0, 1, 2, 3, 4]
```

## 7. Comandos y operadores básicos

- Operadores comunes en cadenas, tuplas, rangos y listas

La siguiente lista de operadores, puede utilizarse en cadenas, tuplas, rangos\* y listas:

- `seq[i]` → *i*º elemento de seq
- `len(seq)` → longitud de seq
- `seq1 + seq2` → concatenación (no para rangos)
- `n * seq` → repetición de una secuencia n veces (no para rangos)
- `seq [inicio:fin]` → porción de la secuencia
- `e in seq` → `True`, si `e` existe dentro de seq
- `e not in seq` → `True`, si `e` NO existe dentro de seq
- `for e in seq` → itera sobre los elementos de seq

## 8. Mutación, alias y clonación de listas

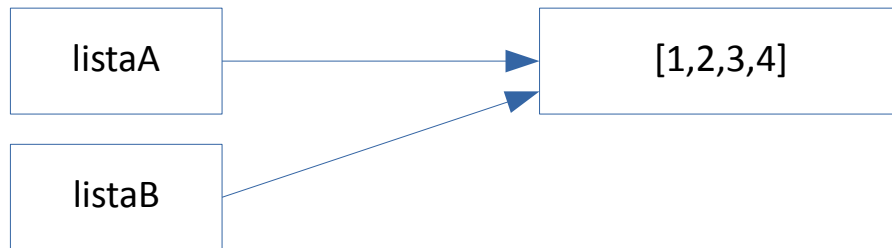
- ¿Qué sabemos de las listas hasta ahora?
  1. Son mutables (pueden cambiar).
  2. Se comportan de manera distinta a los tipos inmutables (tuplas).  
-----
  3. Son objetos que se alojan en memoria.
  4. El nombre de la variable APUNTA al objeto.
  5. **Podemos tener diferentes nombres de variable apuntando al mismo objeto → Si el objeto cambia, también lo hacen todas las variables que apuntan a él.**
- Todo lo anterior puede tener efectos secundarios cuando trabajamos con listas.

## 8. Mutación, alias y clonación de listas

```
listaA = [1,2,3,4]
```

```
listaB = listaA
```

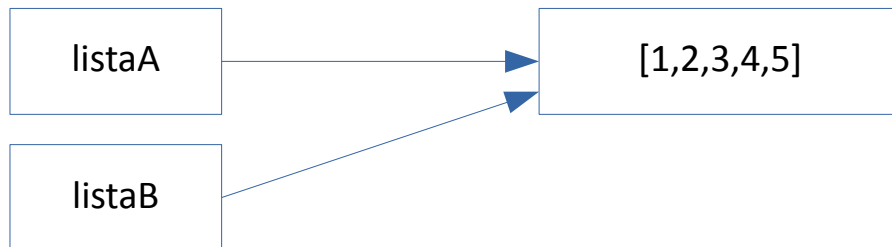
```
listaB → [1,2,3,4]
```



- ¿Qué pasa si añado un elemento a la listaA?

```
listaA.append(5) -> [1,2,3,4,5]
```

```
listaB → devuelve  
[1,2,3,4,5]
```



- Esto se debe a que tanto listaA, como listaB, son enlaces a un objeto en memoria. Si cambiamos dicho objeto en memoria, cualquier enlace que haga referencia al objeto cambia.

## 8. Mutación, alias y clonación de listas

El comportamiento es similar al de los accesos directos en Windows. Cualquier cambio en el objeto original al que apunta un acceso directo, se muestra en cualquiera de los ALIAS.

Sin embargo, si en vez de añadir un elemento a una lista, defino una nueva lista:

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

```
listaB → [1,2,3,4,5]
```

Esto se debe a que listaA ahora apunta a un objeto diferente en memoria al de listaB.

- Puede parecer trivial pero tiene implicaciones importantes cuando estemos programando, ya que un cambio en una lista en un punto del programa, puede tener efectos secundarios no deseados más adelante.

## 8. Mutación, alias y clonación de listas

Supongamos que hacemos:

```
listaA = [1,2,3,4,5]
```

```
listaB = [1,2,3,4,5]
```

En este caso ambas listas son diferentes, ya que apuntan a diferentes objetos en memoria. Los objetos son iguales, sí, pero ubicados en diferentes lugares de la memoria.

```
listaA.append(6)
```

```
listaA → [1,2,3,4,5,6]
```

```
listaB → [1,2,3,4,5]
```

## 8. Mutación, alias y clonación de listas

La **mutación** de una lista, es el cambio (total o parcial) de los elementos de la misma.

```
listaA = [1,2,3,4,5]
```

```
listaA[1] = 1000
```

```
listaA → [1,1000,3,4,5]
```

## 8. Mutación, alias y clonación de listas

Al realizar iteraciones con listas, debemos evitar operaciones que impliquen mutación.

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

```
lista2=[1,2,5,6]
```

```
# eliminar duplicados de una lista
```

```
for a in lista1:
```

```
    if a in lista2:
```

```
        lista1.remove(a)
```

```
→ lista1: [2, 3, 4]
```

```
→ lista2: [1, 2, 5, 6]
```

Vemos que lista1, no es [3,4].



## 8. Mutación, alias y clonación de listas

Esto se debe a que Python usa un contador interno para saber en que posición de la lista está. Si mutamos una lista (en nuestro caso la hemos reducido), cambia su tamaño, PERO no actualiza el contador y por eso muestra [2,3,4]

- Para evitar este efecto secundario podríamos hacer:

```
lista1=[1,2,3,4]
lista2=[1,2,5,6]
lista1_copia = lista1[:] → clona lista1
for a in lista1_copia:
    if a in lista2:
        lista1.remove(a)
```

Cuando muta lista1,  
no lo hace lista1\_copia

→ lista1: [3, 4]

→ lista2: [1, 2, 5, 6]

## 8. Mutación, alias y clonación de listas

- La **clonación** de una lista, permite duplicar una lista en otro espacio de memoria (independiente del original).
- Básicamente es una copia del objeto original.
- La nueva copia es independiente de cualquier cambio que se produzca en el objeto original.

```
listaA = [1,2,3,4,5]
```

```
listaB = listaA[:]
```

→ La clonación nos permite realizar operaciones que implicación mutación en la lista, sin cambiar la lista original.

## 9. Control de flujo - Condicionales

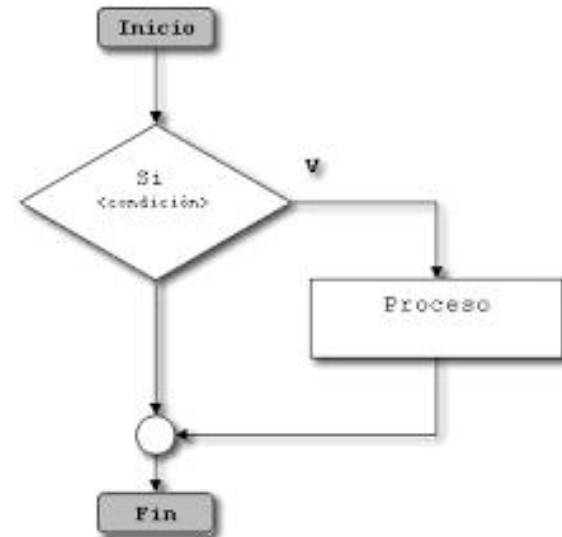
- Un script se ejecuta de manera lineal de principio a fin.
- El control de flujo nos permite romper esta linealidad de manera que podamos saltarnos sentencias dentro del script en función de determinada/s condición/es.

*if condición:*

*órdenes a ejecutar si  
la condición es cierta*

*else:*

*órdenes a ejecutar si  
la condición NO es cierta*



## 9. Control de flujo - Condicionales

Ej:

```
a = 3
b = 4
if a == b :
    print ("a y b son iguales")
else:
    print ("a y b son diferentes")
```

## 9. Control de flujo - Condicionales

El sangrado en el bloque de instrucciones, permite que el intérprete sepa qué instrucciones corresponden a un bloque.

*Ejemplo:*

```
edad = int(input("¿Cuántos años tienes? "))
if edad < 18:
    print("Eres menor de edad")
    print("No puedes conducir coches todavía")
else:
    print("Eres mayor de edad")
    print("Enhorabuena, puedes sacarte el carnet de conducir")
print("¡Hasta la próxima!")
```

## 9. Control de flujo - Condicionales

Podemos encontrarnos situaciones en las que no tengamos solo 2 opciones.

if condición a evaluar 1:  
    sentencias a ejecutar

elif condición a evaluar 2:  
    sentencias a ejecutar

else:  
    sentencias a ejecutar



Condición ejecutada por descarte

## 9. Control de flujo - Condicionales

Las condiciones puede anidarse, de manera que dentro de una condición podemos incluir otras condiciones:

```
if condición a evaluar 1:
    sentencias a ejecutar
    if condición a evaluar 1.2:
        sentencias a ejecutar
    else:
        sentencias a ejecutar
elif condición a evaluar 2:
    sentencias a ejecutar
else: condición a evaluar 3:
    sentencias a ejecutar
    if condición a evaluar 3.1:
        sentencias a ejecutar
    else:
        break
sentencias a ejecutar
```

## 10. Iteraciones

Las iteraciones nos permiten ejecutar determinadas porciones de código de manera repetitiva.

Son muy útiles porque permiten simplificar nuestro código y que éste sea más legible.

Existen 2 tipos de iteraciones en Python:

- While
- For

- Ambos tipos de iteraciones, aunque permiten repetir una porción de código, funcionan de manera diferente.



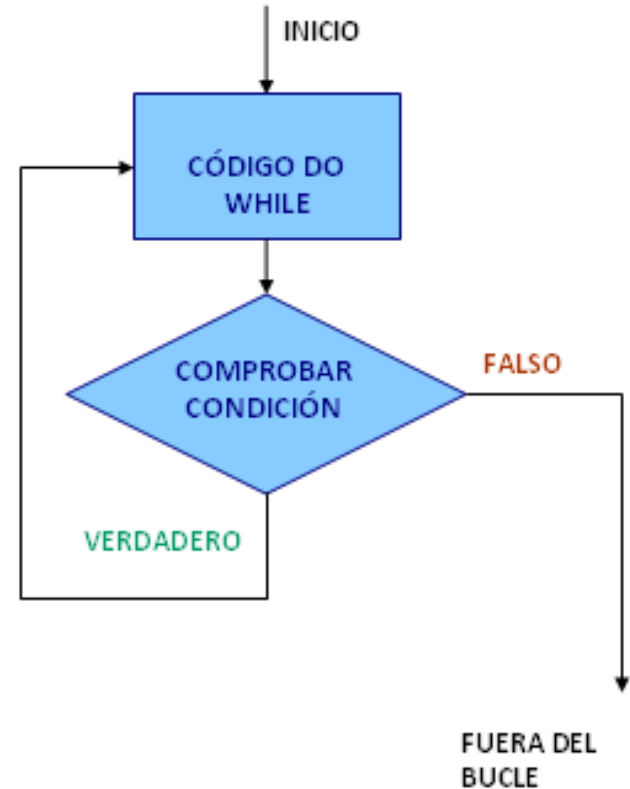
## 10. Iteraciones

- while

Permite repetir la ejecución de un grupo de instrucciones, mientras se cumpla una condición.

¡OJO! El bucle while, si no está bien definido, nos puede llevar a un **bucle infinito**.

La variable o variables que aparezcan en la condición (variables de control), deben definirse antes del bucle y modificarse en el bucle para no terminar en un bucle infinito (contador).



## 10. Iteraciones

- while

Sintaxis:

*while condición:*  
*cuerpo del bucle*

Ejemplo: Imprimir una secuencia de números, que empezando en 0, no supere el 10.

```
a = 0
while a < 10:
    print (a)
    a += 1
```

## 10. Iteraciones

- while

Ejemplo:

Supongamos el siguiente juego sencillo.  
Consiste en salir de un callejón, donde sólo podemos ir a la izquierda o derecha.  
Si vamos a la izquierda, salimos del callejón, si vamos a la derecha seguimos perdidos.

En este ejemplo, mientras se pulse derecha, el programa continúa indefinidamente.

```
direccion = input("Estás perdido en el callejón. ¿Izquierda o Derecha (I/D)?")
while direccion == "D":
    direccion = input("Estás perdido en el callejón. ¿Izquierda o Derecha (I/D)?")

print ("Has conseguido salir del callejón")
```

Estás perdido en callejón

\*\*\*\*\*

\*\*\*\*\*



\*\*\*\*\*

\*\*\*\*\*

¿Izquierda o derecha para salir del callejón?

## 10. Iteraciones

- while

Al usar while, es conveniente el **uso de variables de control**.

```
n = 0
while n < 5:
    print(n)
    n = n + 1
```

¿Qué pasa si no definimos `n = 0`? → Se genera un error, ya que la variable `n` no está definida.

¿Qué pasa si eliminamos la línea `n = n + 1`? → Entramos en un bucle infinito.

## 10. Iteraciones

- while y el manejo de excepciones

Python maneja el uso de excepciones (errores detectados durante la ejecución de un programa) a través de:

```
try:
    <sentencias a ejecutar>
except Excepción_A_Controlar:
    <sentencias a ejecutar si try: es TRUE>
```

- Relación de excepciones que podemos controlar en Python:

<https://docs.python.org/3/library/exceptions.html#builtin-exceptions>

Existe prácticamente un “Exception handler” para cada tipo de excepción existente en Python.

## 10. Iteraciones

- while y el manejo de excepciones

Ej: Programa que espera que tecleemos un número y maneja la excepción en caso de no hacerlo.

```
while True:
    try:
        x = int(input("Introduce un número: "))
        break
    except ValueError: #Excepción a manejar de tipo valor.
        print("¡Vaya! Eso no era un número válido. Prueba otra vez...")
```

## 10. Iteraciones

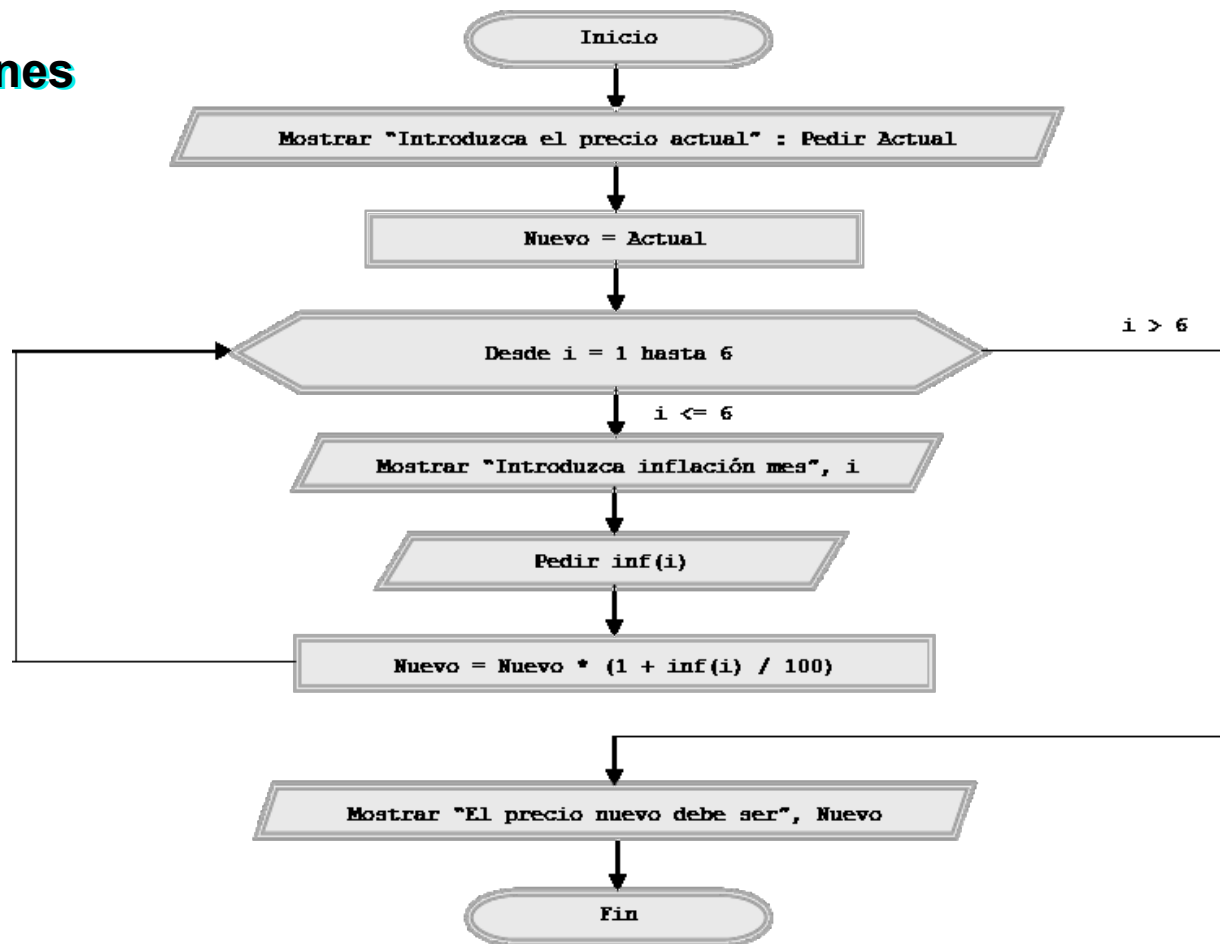
- for

Para eliminar los problemas mencionados con While, podemos usar **For**. Es otro tipo de iteración y permite repetir la ejecución de un grupo de instrucciones un número **DETERMINADO** de veces.

- No es necesario definir variables de control, ya que la ejecución del bucle For, ES FINITA y definida.
- En cada iteración, el bucle For, suma internamente X al contador (generalmente se suma 1, pero este valor lo podemos definir).
- Algunos tipos de iteraciones pueden realizarse indistintamente con FOR y WHILE (generalmente toda iteración que pueda ejecutarse con for puede realizarse con while, pero no al revés).

## 10. Iteraciones

- for





## 10. Iteraciones

- for

Sintaxis:

```
for variable in (lista, cadena, range, ...):  
    cuerpo del bucle
```

Ejemplo:

```
for a in range (1,10):  
    print (a)
```

```
b = 0
```

```
for a in range (1,10):  
    b += a  
    print ("Iteración " + str(a) + ", valor de b:" + str(b))
```

## 10. Iteraciones

- for

El rango del bucle **for** puede empezar y terminar en cualquier número entero.

```
for a in range (100,150):
```

Los bucles **for** también pueden ser decrecientes:

```
for a in range (150, 100, -5):
```

¡OJO! El último elemento del rango NO se evalúa. En el último ejemplo, el bucle pararía en el caso inmediatamente anterior a 100 (105).

## 10. Iteraciones

- for anidado

Podemos anidar bucles dentro de bucles para realizar repeticiones dentro de las iteraciones.

- *Ej: Supongamos que tenemos una matriz de datos de 10 x 8 (matriz). En cada celda hay un valor numérico. Podríamos utilizar un bucle anidado para poder sumar el valor de todas las celdas. Para ello necesitamos crear un sistema que lea secuencialmente los valores de la tabla uno a uno (celda a celda).*

```
suma = 0
for filas in range (10):
    for columnas in range (8):
        suma += matriz [filas,columnas]
```

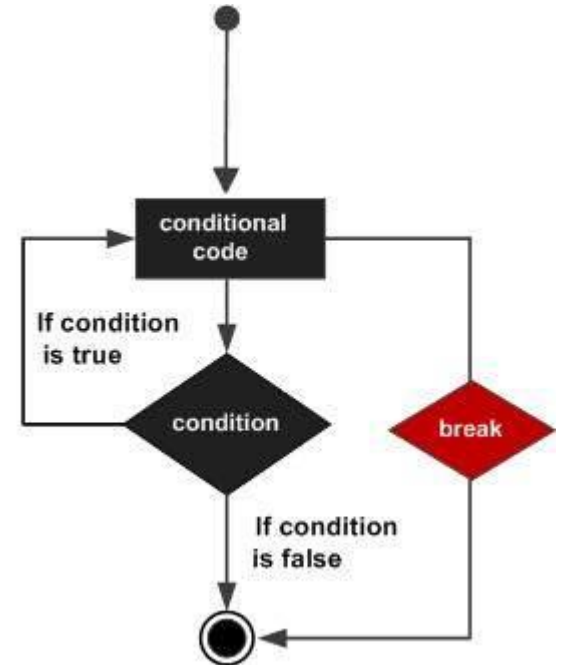
## 10. Iteraciones

- break

Durante la ejecución de las iteraciones (y condicionales), puede interesarnos cancelarlas bajo determinadas condiciones.

P.ej: Podríamos crear un código muestre algo en pantalla hasta que pulsemos una tecla y cuando lo hagamos muestre otra cosa.

Puede usarse tanto con **FOR** como con **WHILE** e **IF**



## 10. Iteraciones

- **break**

Ejemplo:

```
suma = 0
for i in range (5, 11, 2):
    suma += i
    if suma == 5:
        break
print (suma)
```

En el caso de ejecutar break dentro de loops (for o while), break termina la ejecución de la iteración actual.

Ejemplo:

```
var = 10
while var > 0:
    print ('Valor actual de la variable :', var)
    var = var -1
    if var == 5:
        break

print ("Adios!")
```

## 10. Iteraciones

- continue

El comando **continue**, salta a la siguiente iteración de un bucle.

Ejemplo:

```
var = 10
while var > 0:
    var = var -1
    if var == 5:
        continue
    print ('Valor actual de la variable :', var)

print ("Adios!")
```



```
Valor actual de la variable : 9
Valor actual de la variable : 8
Valor actual de la variable : 7
Valor actual de la variable : 6
Valor actual de la variable : 4
Valor actual de la variable : 3
Valor actual de la variable : 2
Valor actual de la variable : 1
Valor actual de la variable : 0
Adios!
```

Cuando `var = 5` → salta la línea que imprime el valor actual de la variable.

## 10. Iteraciones

- Resumen

for	while
Usar cuando conozcamos el total de iteraciones a ejecutar (rango de ejecución)	Usar cuando desconocemos el total de iteraciones a ejecutar.
Podemos cancelar la ejecución a través de <b>break</b>	Podemos cancelar la ejecución a través de <b>break</b>
Utiliza un contador interno	Es necesario definir un contador, que debe inicializarse antes de ejecutar el bucle, así como incrementarse (o reducirse) dentro del bucle.
Puede reescribirse para usando <b>while</b>	Puede no ser posible reescribirse usando <b>for</b>

## 10. Iteraciones

- Ejemplo

- ➔ Crear un programa que calcule el cuadrado de un número entero, mediante la suma de si mismo n-veces.

$$2^2 = 2 + 2 \text{ (dos, dos veces)}$$

$$3^2 = 3 + 3 + 3 \text{ (tres, tres veces)}$$

$$4^2 = 4 + 4 + 4 + 4$$

```
x = int (input("Introduce un valor entero y calcularé su cuadrado: "))
cuadrado = 0
iteraciones_pendientes = x
while iteraciones_pendientes != 0:
    cuadrado = cuadrado + x
    iteraciones_pendientes -= 1
    # iteraciones_pendientes = iteraciones_pendientes - 1

print (str(x)+ "*" + str(x) + " = " + str(cuadrado))
```



## 10. Iteraciones

- Ejemplo

Paso a paso a través del código.

Arranque del programa

Inicio del bucle

Fin del bucle, cuando  
iteraciones\_pendientes = 0

Valor de x	cuadrado	iteraciones_pendientes
3	0	3
	3	2
	6	1
	9	0
4	0	4
	4	3
	8	2
	12	1
	16	0

## 10. Iteraciones

- Ejemplo

Como vemos en el ejemplo, para ejecutar correctamente el programa, necesitamos:

- 1) Definir fuera del bucle, la variable usada para la iteración (iteraciones\_pendientes) → variable test
- 2) Comprobar la variable test para determinar cuándo hemos terminado con el bucle.
- 3) Modificar la variable test dentro del bucle para no acabar en un bucle infinito.

## 10. Iteraciones

- Ejemplo

Reescribir el programa anterior para que ejecute el proceso a través de un **for**

## 10. Iteraciones

- Guess and check

Conociendo como aplicar iteraciones y condiciones para encontrar la solución a determinados problemas mediante el **método de prueba y error**.

No es un método muy eficiente (a veces es muy lento).

- ➔ Básicamente necesitaré crear un programa que genere un dato de prueba y comprobar cómo de acertado es ese dato generado. Si el dato no es acertado, generar un nuevo dato y volver a comprobarlo. Seguir repitiendo este proceso, hasta encontrar un valor acertado.

## 10. Iteraciones

- Guess and check

Para entenderlo vamos a escribir un programa que nos calcule la raíz cúbica de un número entero...  
**sin usar la función raíz cúbica.**

La condición a evaluar será, Sí **prueba = valor de x** → hemos encontrado la raíz cúbica de X, donde la raíz corresponde al valor de la iteración.

Valor de x	iteración	prueba
27	0	$0 ** 3 = 0$
	1	$1 ** 3 = 1$
	2	$2 ** 3 = 8$
	3	$3 ** 3 = 27$
125	0	$0 ** 3 = 0$
	1	$1 ** 3 = 1$
	2	$2 ** 3 = 8$
	3	$3 ** 3 = 27$
	4	$4 ** 3 = 64$
	5	$5 ** 3 = 125$

## 10. Iteraciones

- **Guess and check**
  - Crear un programa que calcule para un número entero positivo, su raíz cúbica.
  - Si el número aportado, no tiene un cubo perfecto, el programa no tendrá que indicar explícitamente.
  - Si tiene cubo perfecto, mostrar el valor de su raíz cúbica.

## 10. Iteraciones

- Guess and check

```
x = int(input('Introduce un número entero: '))
for prueba in range (x+1):
    if prueba ** 3 == x:
        print ('La raíz cúbica de', x,'es',prueba)
```

¿Veis algún fallo?

Si  $x = 27$ , ¿el programa hace lo que se espera de él?

¿Si  $x = 125$ ?

¿Si  $x = 28$ ?

¿Si  $x = -27$ ?

## 10. Iteraciones

- Guess and check

- Vemos como el programa anterior solo funciona para números positivos.
- El rango de actuación del programa, ¿debe circunscribirse, solo al ámbito de los números positivos?, o ¿debería incluir también los números negativos?.
- ¿Nuestro programa está considerando todas las opciones posibles?
- Cuando vayamos a crear un programa, debemos tener en cuenta TODAS las opciones que pueden existir para que la ejecución del programa las tenga en cuenta.
- Para el ejemplo anterior, un programa que calcule la raíz cúbica de un número entero tendría que evaluar una condición adicional (si el número es negativo o no).



## 11. Funciones

- A medida que nuestros programas aumentan en complejidad y acumulan líneas de código, se hace cada vez menos fácil poder mantenerlos.
- Debemos tender a realizar código lo más limpio y simple posible. Ésto cobra especial relevancia si nuestro código se va a ejecutar en la nube, donde se paga por tiempo de ejecución.
- No es mejor un programa con mucho código, sino aquel que con el mismo código tiene mayor funcionalidad.
- Esto nos lleva a las **funciones**.

## 11. Funciones

- Las funciones, nos permiten “encapsular” porciones de código y se basan en:
  1. Descomponer un problema en partes más simples (también llamado modularidad) que contienen todo lo necesario para ejecutar su tarea.
  2. Abstracción (no necesito saber cómo funciona algo, solo qué hace).
- Podemos ver las funciones como programas que se ejecutan dentro de un programa.
- No necesitamos saber cómo funciona un programa, mientras sepamos como interactuar con él.

## 11. Funciones

*Hoy en día, prácticamente todo el mundo sabe conducir, pero ¿cuántos conocen la física que rige el funcionamiento de un motor?*

*Una televisión, ¿cuántos conocemos cómo funciona la electrónica que hace funcionar una televisión?. Sabemos cómo “manejarla” = interactuar y no necesitamos más. Esto es lo que se conoce como ABSTRACCIÓN.*

*La descomposición del problema en partes más simples, ayuda a resolverlo con mayor eficiencia. Un problema grande puede resolverse, resolviendo pequeñas partes. En el caso de la televisión, la misma no se fabrica de una vez, se fabrica por partes que luego se ensamblan y en conjunto funcionan como una sola parte.*

## 11. Funciones

Por tanto, podemos ver las funciones como CAJAS NEGRAS a las que enviamos unos valores y nos devuelven un resultado.

- Para conocer qué hace una función, es necesario documentarla (en el propio código) como veremos un poco más adelante.
- Una vez que hemos definido una función, y comprobado que funciona, funcionará siempre que la alimentemos con datos para los que está diseñada.
- Una función no se ejecuta, hasta que sea llamada o invocada dentro de un programa.

## 11. Funciones

Podemos pensar en una función como una receta de cocina. Supongamos que queremos hacer un bizcocho con nata recubierto con chocolate.

Dicha receta, podríamos dividirla en 3 partes (Descomposición en partes más simples):

- Hacer el bizcocho
- Montar la nata
- Preparar la cobertura.

Ahora que tenemos definidas 3 partes más simples, podemos hacer una receta para cada paso, de manera que una vez que yo haya hecho un bizcocho correctamente, ya sé que dicha receta es buena (Abstracción).

## 11. Funciones

- Características de una función:
  - ➔ Tiene un nombre
  - ➔ Tiene parámetros (0 ó más)
  - ➔ Contiene un docstring (opcional) → documentación que nos dice lo que hace la función.
  - ➔ Tiene un cuerpo. → Secuencia de comandos que se ejecutan en la función.

# 11. Funciones

- Características de una función:

Palabra clave  
para definir  
una función

Documentación sobre  
la función, lo que hace,  
necesita y devuelve

```
def es_par (i):
```

```
    """
```

```
    Created on Sun Apr 22 18:19:00 2018
```

```
    @author: ABC
```

```
    Entrada: i, número entero positivo.
```

```
    Salida: Devuelve True si el número es par, de otra manera devuelve False
```

```
    """
```

```
    print ("Hola")
```

```
    return i%2 == 0
```

Tabulación  
(Indentación)

Expresión  
a evaluar

Cuerpo de  
la función

Nombre de  
la función

Parámetros

## 11. Funciones

- **Ámbito de las variables**

Podemos entender “ámbito” de una variable como el **entorno** donde trabaja una variable (también llamado “marco de trabajo”).

Cuando arranco un programa, se genera un entorno con las variables del mismo.

- ➔ Si dentro de dicho programa, llamamos a una función, las variables de dicha función **EXISTIRÁN EN SU PROPIO** espacio de trabajo, **QUE ES DIFERENTE** al del programa que ha llamado a la función.



## 11. Funciones

- **Ámbito de las variables**

Supongamos la siguiente función:

```
def suma_uno(x):  
    x = x + 1  
    print ('Dentro de suma_uno(x), x =', x)  
    return x
```

- Asignamos un valor a x=3

```
x = 3  
z = suma_uno(x)
```

- La función nos devuelve

```
Dentro de suma_uno(x), x = 4
```

- Si consultamos el valor de x, nos devuelve 3

- ➔ ¿Qué está pasando? ¿Porqué dentro de la función x=4 y fuera x=3, si parecen ser la misma variable?

## 11. Funciones

- **Ámbito de las variables**

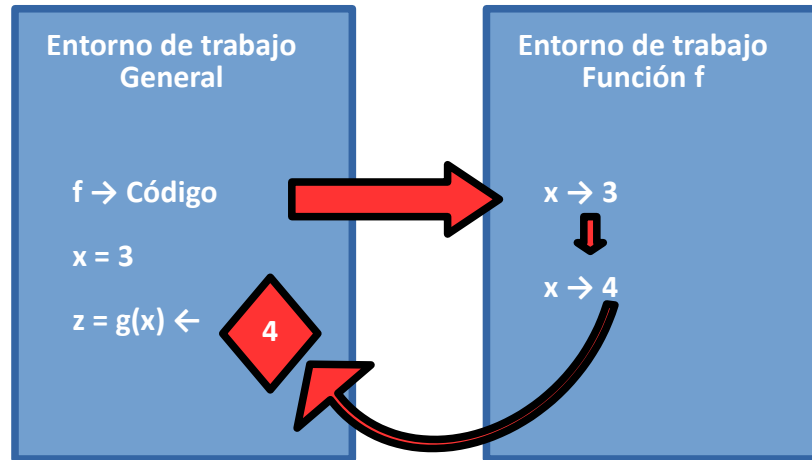
Básicamente, lo que está pasando es que el ámbito de la variable `x` es DIFERENTE al de la variable `x` contenida dentro de la función.

- ➔ Al pertenecer ambas variables a diferentes ámbitos, SON 2 variables diferentes para Python.
- ➔ Los valores de las variables dentro del marco de trabajo de una función, se destruyen una vez finalizada la ejecución de dicha función.

# 11. Funciones

- Ámbito de las variables

```
def suma_uno(x):  
    x = x + 1  
    print ('Dentro de suma_uno(x), x =', x)  
    return x
```



# 11. Funciones

- **Ámbito de las variables**

¿Qué pasa si nuestra función no incluye return? Python devuelve el valor None (ausencia de valor) y por tanto perdemos cualquier cálculo realizado dentro de la función.



Una vez que se termina de ejecutar una función, se destruye su ámbito de trabajo y el valor de todas la variables que pertenecían a dicho ámbito.

Dentro de una función se puede acceder a variables definidas en el ámbito “padre”.

Dentro de una función NO SE PUEDEN modificar variables definidas fuera de su ámbito de trabajo.

## 11. Funciones

- **Ámbito de las variables**

<i><b>def f(x):     x = 1     x += 1     print (x)</b></i>	<i><b>def g(x):     print (x)     print (x + 1)</b></i>
x = 5 f(x) print (x)	x = 5 g(x) print (x)
f(x) = f(5) = 2	g(x) → imprime 5 y 6
print (x) = 5 ← Valor de x en el marco superior	print (x) = 5 ← Valor de x en el marco superior

## 11. Funciones

- return vs. print

return	print
Solo puede usarse dentro de una función.	Puede usarse dentro y fuera de las funciones.
Solo puede ejecutarse 1 return dentro de una función (puede haber múltiples return, pero sólo uno se ejecutará).	Puede ejecutarse múltiples veces dentro de una función.
Cualquier código después de un return NO se ejecuta.	Cualquier código después de un print, puede ejecutarse.
Devuelve un valor a la función que lo llamó.	Devuelve un valor a la consola, pero no se almacena.

# 11. Funciones

- Como argumentos

Una función puede tener como argumento otra función.

Ejemplo:

```
def func_a():  
    print ("Dentro de la función a")  
  
def func_b(y):  
    print ("Dentro de la función b")  
    return y  
  
def func_c(z):  
    print ("Dentro de la función c")  
    return z() # devuelve el valor de z que aquí es una función
```

*Instrucción*

`print (func_a()) →`

`print (5 + func_b(2)) →`

`print (func_c(func_a)) →`

*Devuelve*

Dentro de la función a  
None

Dentro de la función b  
7

Dentro de la función c  
Dentro de la función a  
None

## 11. Funciones

- Como argumentos

Una función también puede contener internamente otra función u otras.

En estos casos, cada función llamada (invocada) desde la función principal, generará su propio entorno de trabajo, con sus propias variables como en los casos previos que hemos visto.



## 11. Funciones

- Como objetos

Las funciones:

- A) Tienen tipos.
- B) Pueden ser elementos de estructuras de datos como listas (¡Importante!).
- C) Pueden asignarse a algo (generalmente una variable).
- D) Pueden usarse como argumento en otra función.

- Son por tanto muy versátiles.

## 11. Funciones

- Como objetos

Ej. Transformar una lista aplicando una función.

```
def transformar_lista (lista,funcion):  
    """ lista: es una lista de datos  
        funcion: es una función que  
        transforma cada dato de la  
        lista  
    """  
    for i in range(len(lista)):  
        lista[i] = funcion(lista[i])
```

- Podemos ejecutar esta transformación, porque las listas son mutables. No podríamos aplicar esta transformación a una tupla o a una cadena, pej.

## 11. Funciones

- Como objetos

Las funciones también pueden ser listas de funciones.

Podemos definir una lista, donde cada elemento sea una función (predefinida o creada por nosotros).

Posteriormente, usar dicha lista en una función y realizar las transformaciones pertinentes.

- Ej: Función que aplica una lista de funciones a un número:

```
def aplicar_funciones (lista,numero):  
    for funcion in lista:  
        print (funcion(numero))
```

```
lista = [abs,int]  
aplicar_funciones (lista,-3.5)  
3.5  
-3
```

## 11. Funciones

- Como objetos

¿Qué pasa si probamos la función previa así?

```
aplicar_funciones (lista, [1,2,3,4,5])
```

```
TypeError: bad operand type for abs(): 'list'
```

Vemos que no podemos aplicar **abs** directamente sobre una lista, debemos aplicar **abs** sobre cada elemento de la lista por separado.

## 11. Funciones

- Valores por defecto

Cuando creamos una función, podemos definir el valor por defecto de sus variables.

Por ejemplo, si en una de nuestras funciones uno de sus valores raramente cambia, podría interesarnos que el programa tome un valor sin que tengamos que proporcionárselo.

Para incluir un valor por defecto:

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

- Ahora la función, devolverá True por defecto, a menos que le proporcionemos un valor.

## 11. Funciones

- **Documentación de las funciones**

Aunque es opcional, es una buena práctica al programar, es documentar en la propia función:

- 1) Condiciones a cumplir por parte del usuario de la función, normalmente se refiere a los valores de los parámetros a usar.
- 2) Resultado obtenido si se cumple el punto 1.

- Esta documentación de lo que hace una función, se incluye en el llamado “docstring” y se coloca al principio de la función delimitado por triples dobles comillas( `""" ... """`).
- Aunque su uso puede parecer poco importante, es de gran ayuda para poder reutilizar código que hayamos creado previamente y del que no recordemos cómo funciona o qué parámetros y de qué tipo necesita.

## 11. Funciones

- Documentación de las funciones

```
def es_par (i):
```

```
    """
```

```
    Created on Sun Apr 22 18:19:00 2018
```

```
    @author: ABC
```

```
    Entrada: i, número entero positivo
```

```
    Salida: Devuelve True si el número es par, de otra manera devuelve False
```

```
    """
```

```
    print ("Hola")
```

```
    return i%2 == 0
```

# 11. Funciones

- Funciones anónimas

Se caracterizan por no utilizar def para su creación.

```
suma = lambda x,y: x+y
```

```
suma (10,20)
```

```
30
```

```
suma (20,20)
```

```
40
```



## 12. Recursión

- **Definición:**

En ciencias de computación, recursión, es una forma de atajar y solventar problemas. Resolver un problema mediante recursión significa que la solución depende de las soluciones de pequeñas instancias del mismo problema.

- La mayoría de los lenguajes de programación dan soporte a la recursión permitiendo a **una función llamarse a sí misma desde el propio programa**.
- Permite realizar iteraciones dentro de un programa SIN utilizar un FOR o un WHILE.
- Es una forma de programación con mucha potencia, para determinados problemas.

## 12. Recursión

- Definición:

Se basa en el principio de “divide y vencerás”, al reducir un problema grande en varios pequeños.

- A la hora de usar esta técnica, deberemos asegurarnos de **no caer en un bucle infinito** (habitualmente incluyendo alguna condición dentro de la función que la finalice).

Veamos cómo...

## 12. Recursión

- **Funcionamiento:**

La idea de solucionar un problema de forma recursiva consiste en dividir el problema en pequeñas partes.

$$a * b = a + a + a + \dots + a \text{ (b veces)}$$

$$= a + (a + a + \dots + a) \rightarrow (b-1 \text{ veces})$$

$$= a + a * (b - 1) \quad \text{Hemos simplificado el problema inicial}$$

$$= a + a + a * (b - 2)$$

Podríamos seguir dividiendo el problema hasta llegar a un punto en el que:

$$b = 1 \rightarrow a * b = a$$

Conociendo el valor del último paso de la recursión podemos rehacer el camino hacia delante.

## 12. Recursión

- Funcionamiento:

### ITERACIÓN

```
def multi_iter (n,m):  
    '''  
    n: número  
    m: multiplicador  
    '''  
  
    resultado = 0  
    while m > 0:  
        resultado += n  
        m -= 1  
    return resultado
```

### RECUSIÓN

```
def multi_recur (n,m):  
    '''  
    n: número  
    m: multiplicador  
    '''  
  
    if m == 1:  
        return n  
    else:  
        return n + multi_recur (n, m-1)
```

## 12. Recursión

- **Funcionamiento:**

Otro ejemplo. Cálculo del factorial de un número.

$$n! = n * (n-1) * (n-2) * (n-3) * ... * 1$$

El cálculo del factorial de un número puede fácilmente descomponerse en partes más simples, con un “caso base” final de 1.

Además  $n! = n * (n-1)!$

Ya tenemos todo lo que necesitamos para solucionar el problema de manera recursiva:

- Caso base donde  $n = 1$
- Simplificación del problema  $n! = n * (n-1)!$

## 12. Recursión

- **Funcionamiento:**

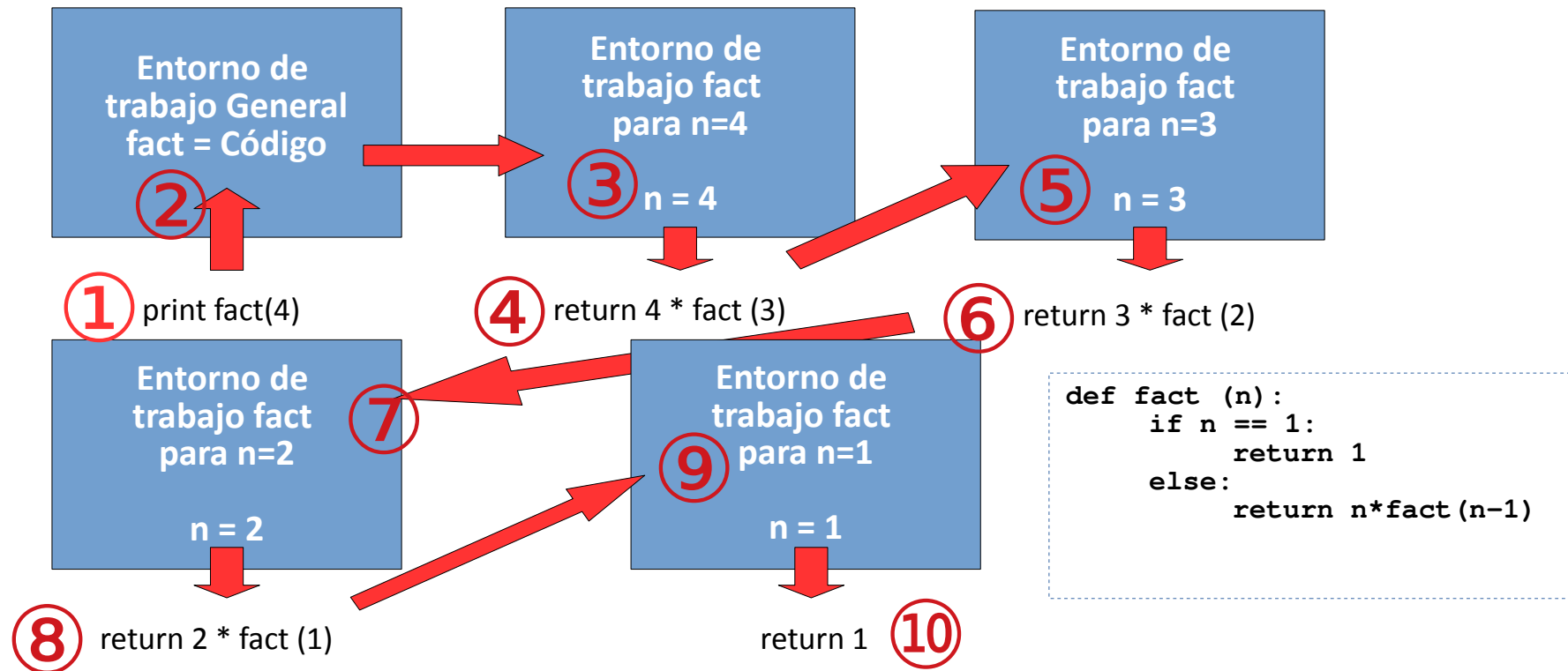
Esas 2 condiciones las podemos resumir en:

```
def fact (n):  
    if n == 1:  
        return 1  
    else:  
        return n*fact(n-1)
```

- ¿Cómo sabe el programa qué valor de n tomar en cada caso y cuándo terminar?. Entra en juego el ámbito de trabajo de las variables que vimos previamente.

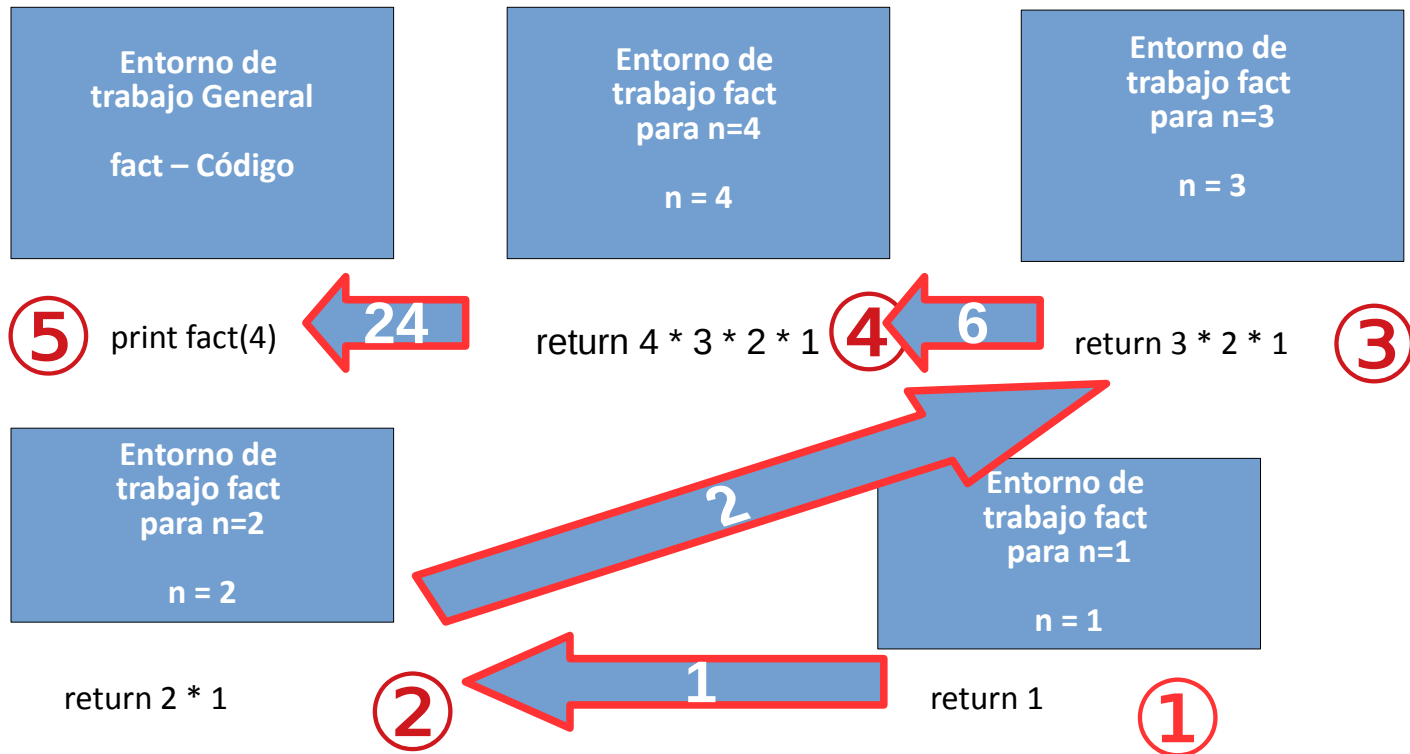
## 12. Recursión

Ida



## 12. Recursión

## Vuelta





## 12. Recursión

Básicamente cada llamada recursiva a una función, crea su propio entorno para las variables.

- Aunque los diferentes entornos compartan variables con el mismo nombre, son variables DIFERENTES y modificar el valor una en un entorno, no modifica el valor en el resto de entornos.
- Resolver problemas mediante recursión, puede ser más eficiente a nivel de programación, pero no tiene porqué serlo a nivel de computación (podemos quedarnos sin memoria en la pila).

## 12. Recursión

### ITERACIÓN

```
def factor_iter(n):  
    resultado = 1  
    for i in range (1,n+1):  
        resultado *= i  
    return resultado
```

### RECURSIÓN

```
def factor_recur(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factor_recur(n-1)
```

## 13. Trabajar con módulos

- **Funcionamiento:**

En Python los archivos que generemos se denominan módulos.

Los módulos tienen la extensión `.py`.

Los archivos `.py` son archivos de texto y pueden editarse con un editor de textos.

Podemos crear módulos con diferentes funciones y guardar todos esos módulos en carpetas.

La agrupación de módulos en carpetas, recibe el nombre de paquete.

- Para que Python entienda que una carpeta es un paquete, debe contener internamente un archivo de texto vacío con el nombre `__init__.py`

## 13. Trabajar con módulos

- **Funcionamiento:**

Cuando la complejidad de nuestros programas aumenta, podemos simplificarlos guardando porciones de código en módulos e invocarlos cuando sea necesario.

- Si sabemos que una porción de código funciona como se espera de él, no hay razón para no utilizarla en otro programa que necesite la misma funcionalidad.

## 13. Trabajar con módulos

- Funcionamiento:

Supongamos que hemos desarrollado las siguientes funciones y comprobado que funcionan bien.

```
pi = 3.14159
```

```
def area (radio):  
    return pi * (radio**2)
```

```
def perimetro (radio):  
    return 2 * pi * radio
```

No necesitamos reescribir o volver a escribir código cada vez que tengamos que calcular el área o el perímetro de una circunferencia.

Por contra, podemos guardar dichas funciones en un módulo (pej: circulo.py).

## 13. Trabajar con módulos

- Funcionamiento:

Para importar un módulo en Python lo haremos a través de: **import**.

Con import, importamos todas las funciones existentes en un módulo

Los módulos a importar deben estar en una ruta que Python sea capaz de encontrar.

```
import circulo
pi = 3
print (pi) -----> Resultado --> 3
print (circulo.pi) -----> 3.14159
print (circulo.area(4)) -----> 50.2654
print (circulo.perimetro(4)) -----> 25.1327
```

## 13. Trabajar con módulos

- Otros metodos de importación:
- Importación selectiva o completa

```
from circulo import * -> Importac. completa  
from circulo import area -> Importac. sel.
```

- A través de la sentencia previa, para invocar el módulo área, solo sería necesario escribir:

```
print (pi) ----- Resultado --> 3.14159  
print (area(4))-----> 50.2654
```

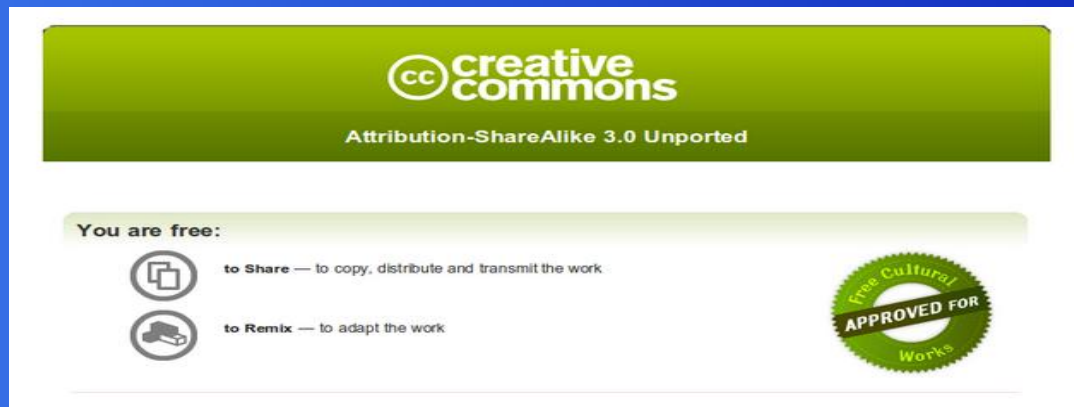
Vemos que en este caso, ya no es necesario usar circulo.area.

Este método solo es válido, si las variables y funciones a importar, no existen en nuestro programa.

## Copyright (c) 2022 Germán Alonso Lascurain

This work (but the quoted images, whose rights are reserved to their owners\*) is licensed under the

Creative Commons “Attribution-ShareAlike” License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/>



Germán Alonso Lascurain

germanalonso@opendeusto.es

galonso@datavaluemanagement.es



Data Value  
management