INICIACIÓN PYTHON
Data BootCamp

Germán Alonso Lascurain
germanalonso@opendeusto.es

TABLA DE CONTENIDOS

1)Introducción
- i) Instalación Python y componentes principales de Spyder (IDE).
- ii)Paquetes incluidos en Anaconda
- iii)Consola/Terminal
- iv)Elementos de un programa Python
- v)Hola Mundo, primer programa.
- vi)Tipos de variables.

TABLA DE CONTENIDOS

2)Operadores básicos

i) Aritméticos
ii) De comparación
iii)De asignación
iv)Lógicos
v) De pertenencia
vi)De identidad
vii)Comandos y operadores básicos
viii)Aritméticos

a) De comparación
b) De pertenencia
c) De identidad
d) Operando con cadenas de texto
e) Operando con listas
f) Operando con tuplas
g) Operando con diccionarios
h) Print
i) Input
j) range()
k) Operadores comunes en cadenas, tuplas, rangos y listas

TABLA DE CONTENIDOS

3)Mutación, alias y clonación de listas
- i) Alias
- ii)Mutación
- iii)Clonación

4)Control de flujo – Condicionales

5)Iteraciones
- i) while
- ii)for
- iii)for anidado
- iv)break
- v)continue
- vi)Resumen
- vii)Ejemplo
- viii)“Guess and check”

TABLA DE CONTENIDOS

6)Funciones

i) Ámbito de las variables

ii)return vs. print

iii)Funciones como argumento

iv)Funciones como objetos

v)Valores por defecto

vi)Documentación de las funciones

7)Recursión

i) Recursión vs. Iteración

ii)Recursión para no numéricos.

TABLA DE CONTENIDOS

8)Trabajar con módulos y archivos
- i) Importar módulos
- ii)Trabajar con ficheros
  - a)CSV
  - b)Ficheros Excel

9)Introducción a la visualización de datos
- i) ¿Porqué visualizar los datos?
- ii)Estructura de un gráfico estadístico
- iii)Pylab

INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

•IDE = Integrated Development Enviroment = Entorno de desarrollo integrado

•Un IDE es un programa que nos ayuda a crear programas.

•Un IDE consta principalmente de:

1)Editor de código.

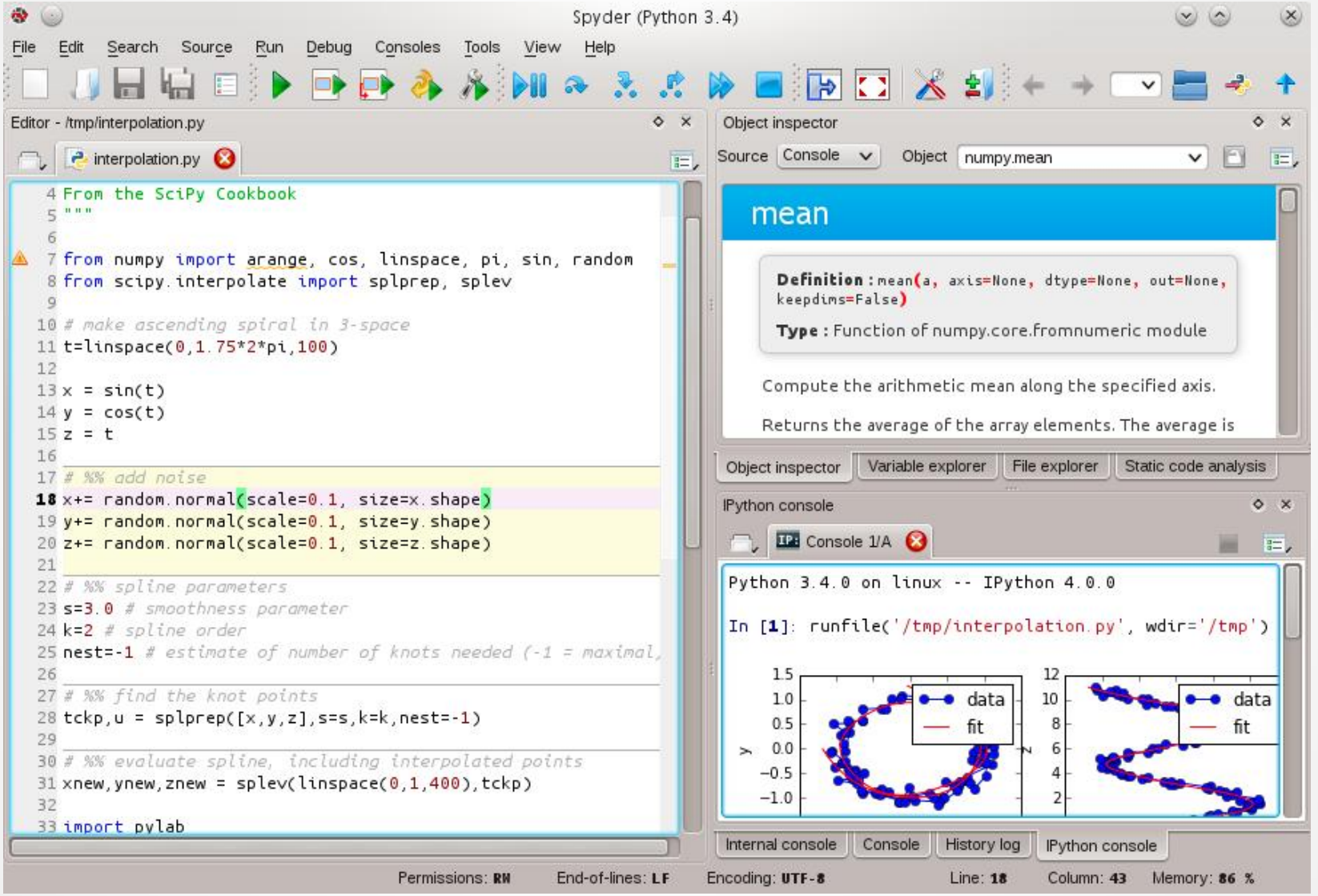
2)Compilador o traductor de código.

3)Un depurador para encontrar errores de programación.

4)Constructor de interfaz gráfica

•Algunos ejemplos: Spyder2, Rstudio, eclipse, android studio, ...

INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER



INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

•Podemos instalar Python “a mano” o bien mediante la ayuda de una de las siguientes distribuciones (en Windows):

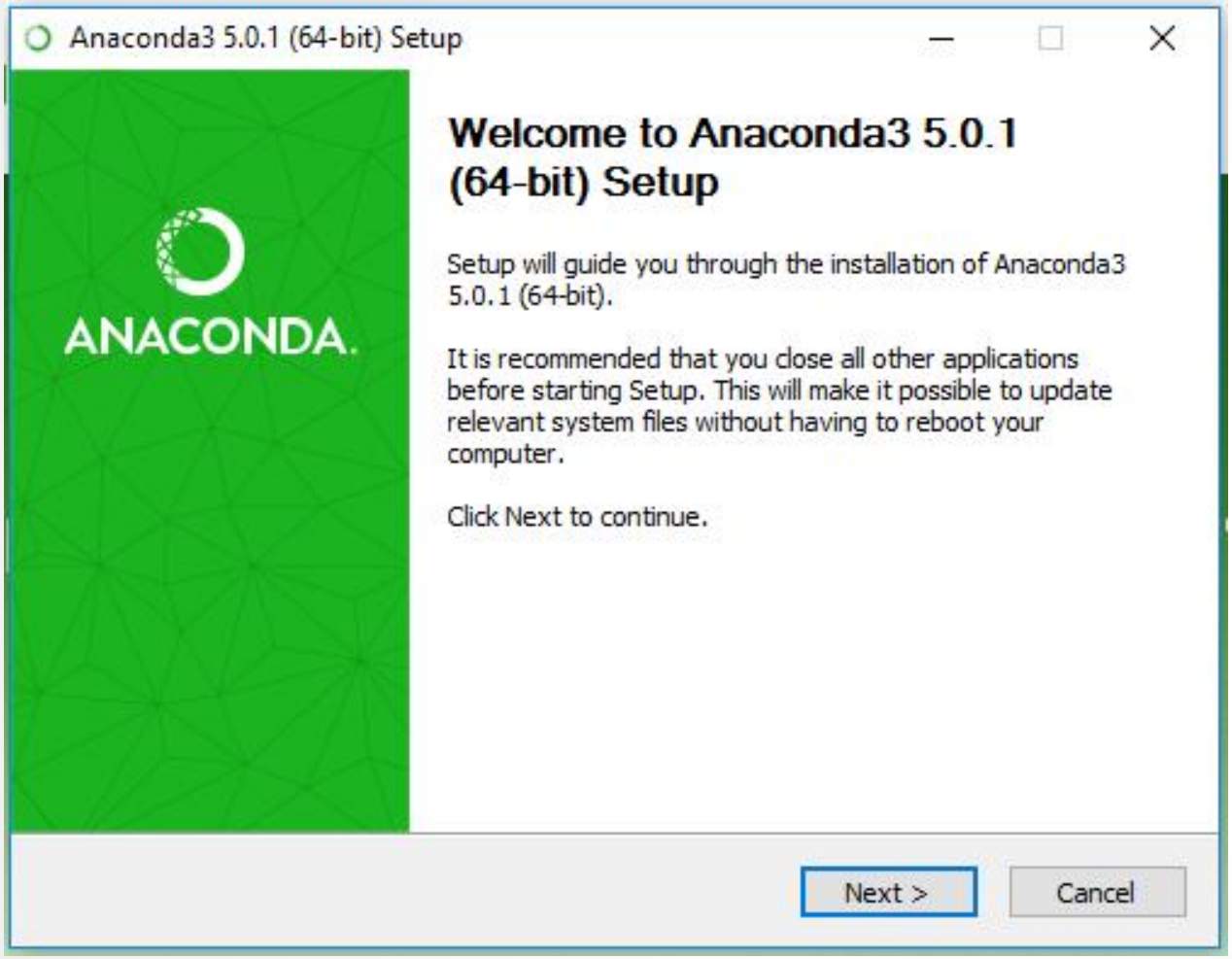
1)Anaconda
2)WinPython
3)Python (x,y)

•Las principales distribuciones Linux ya incorporan Python por defecto, por lo que no sería necesario instalarlo, pero Anaconda simplifica muchísimo el mantenimiento y actualización de los paquetes instalados.

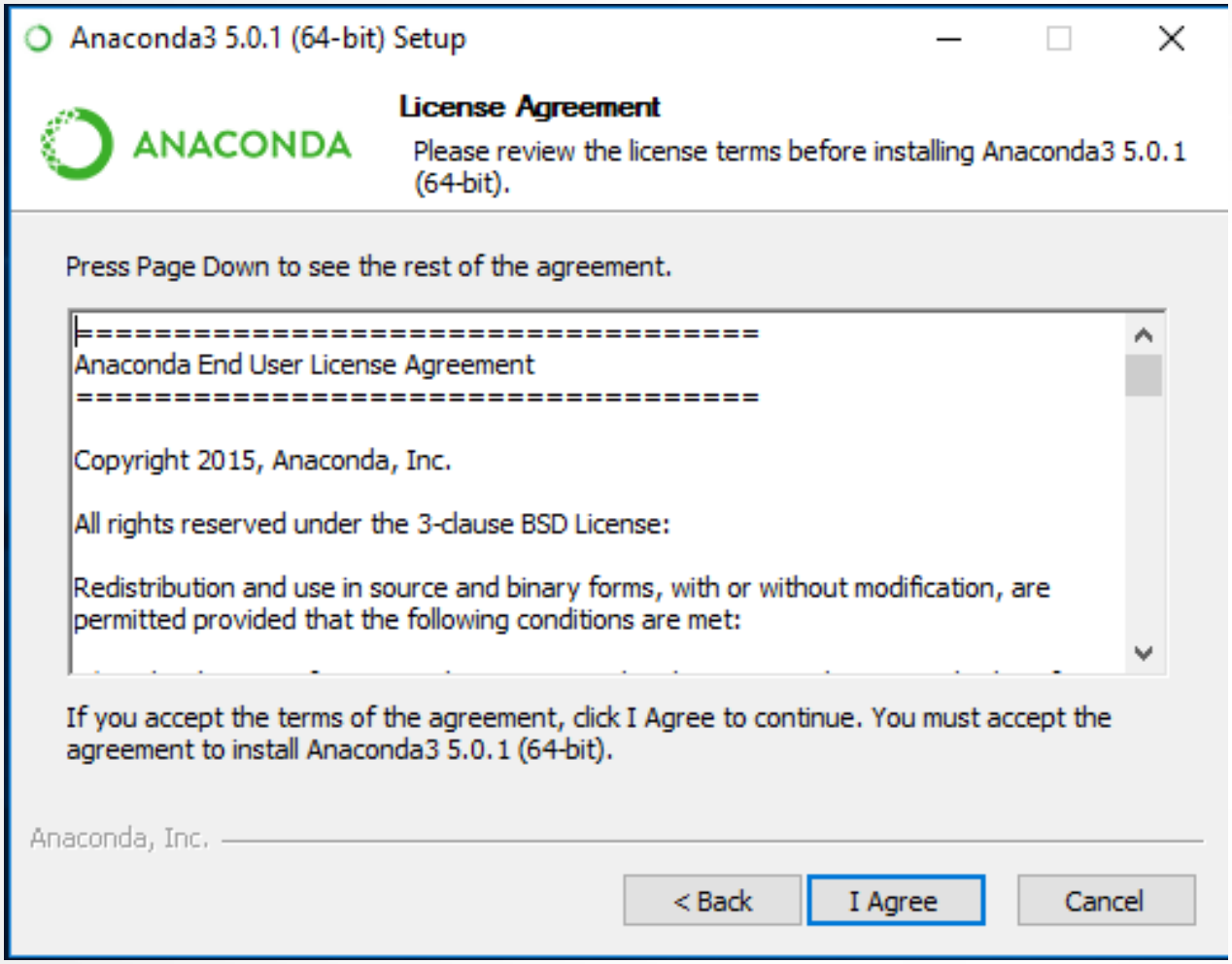
•Nosotros instalaremos Python a través de Anaconda:

https://www.anaconda.com/download

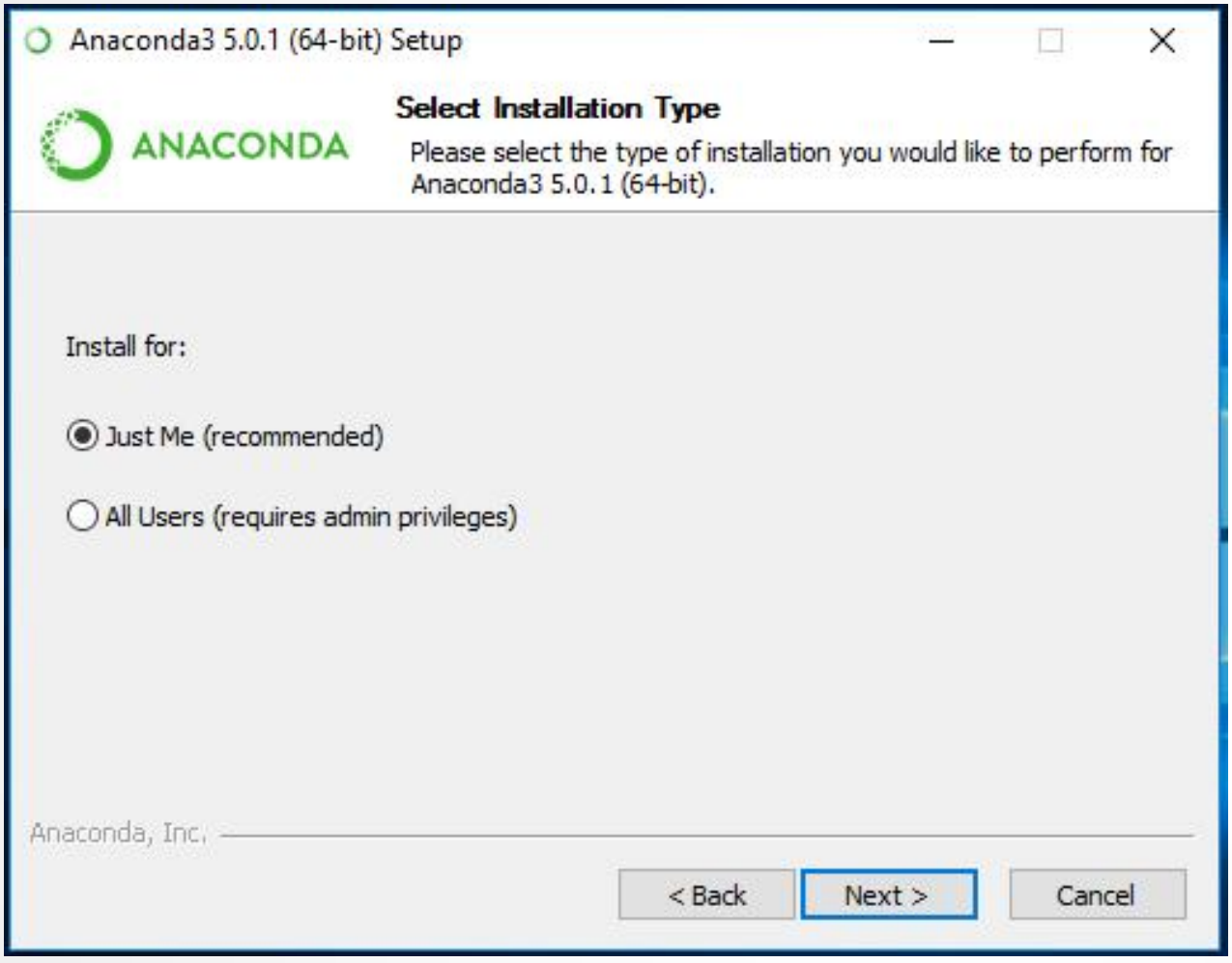
INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER



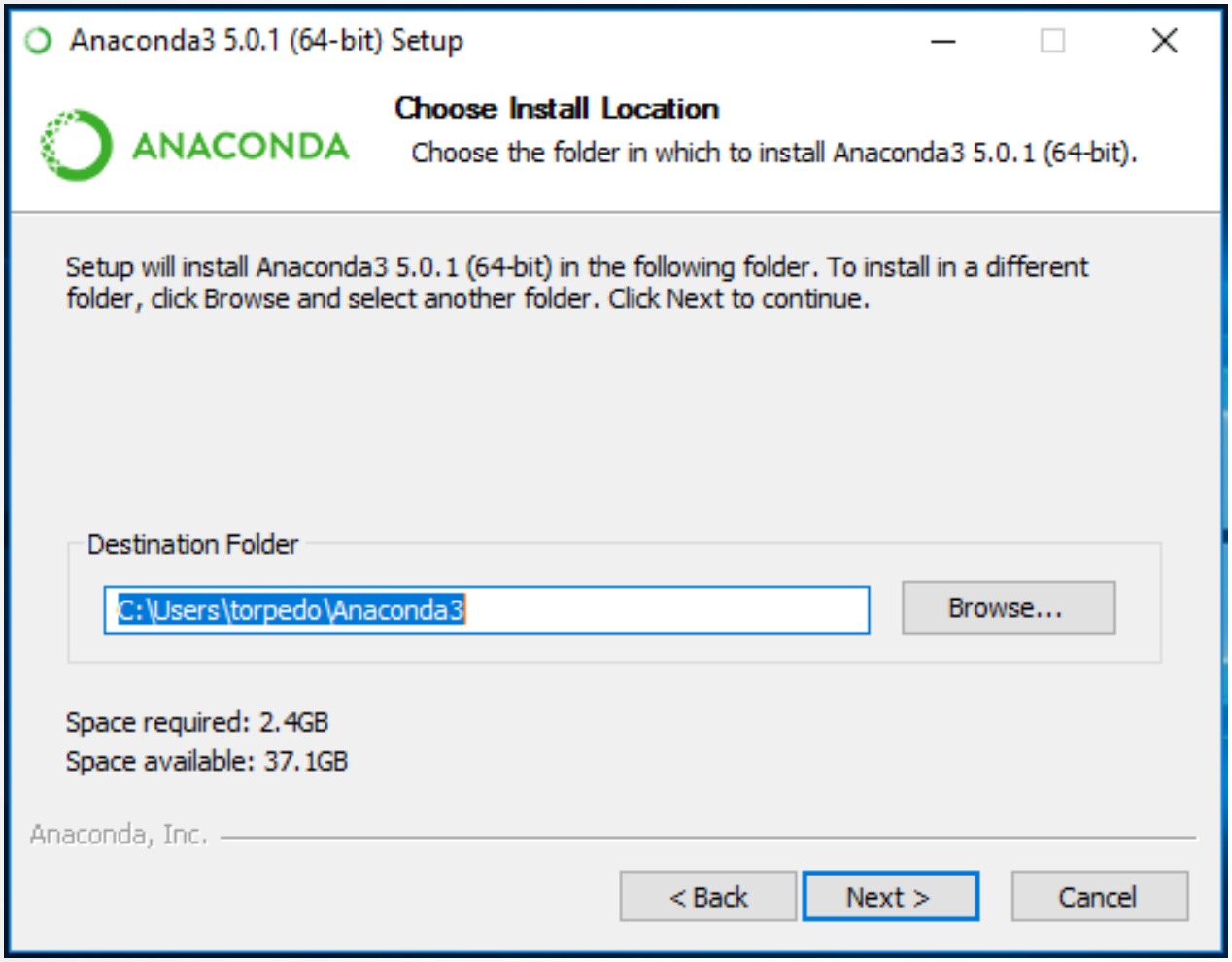
INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER



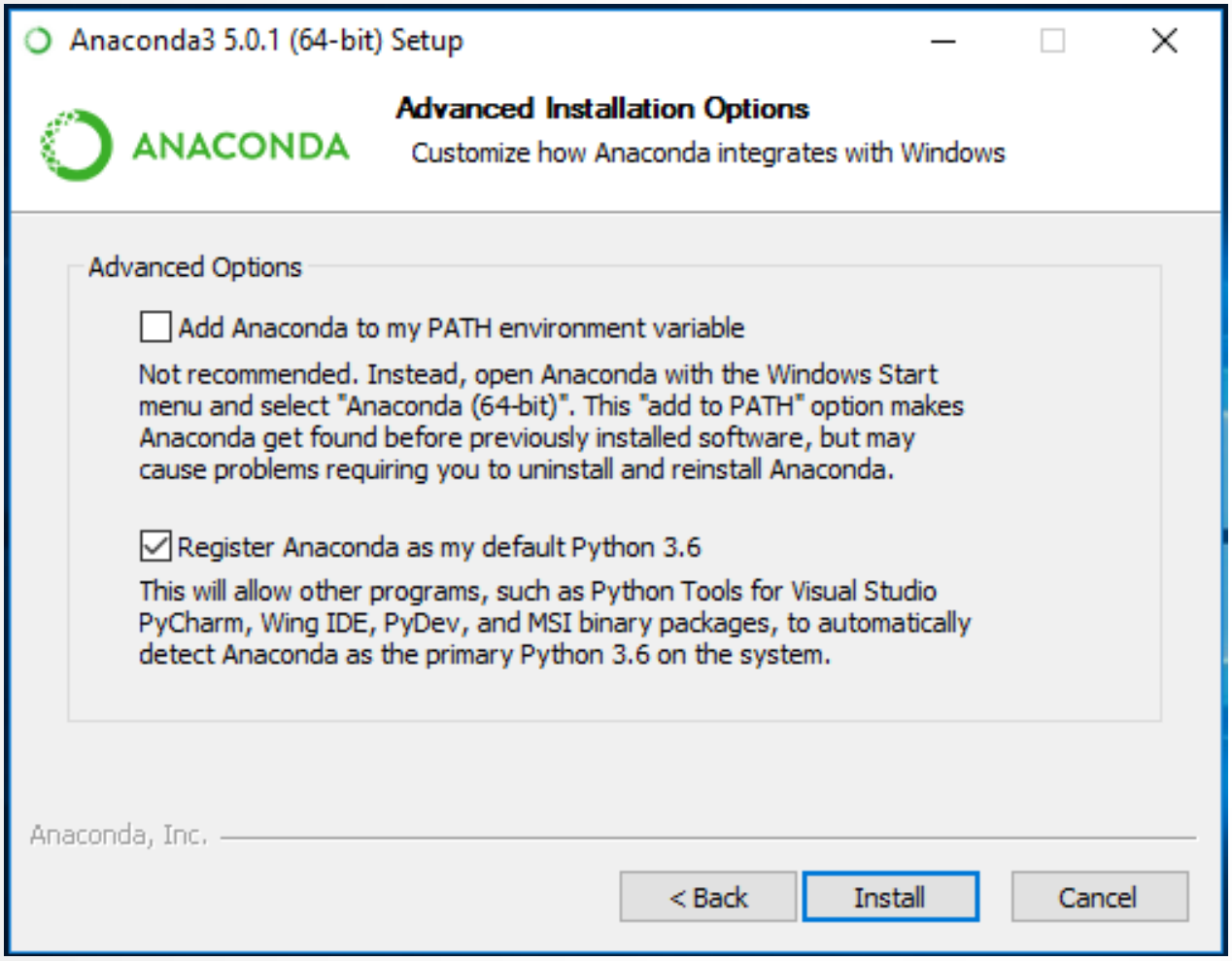
INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER



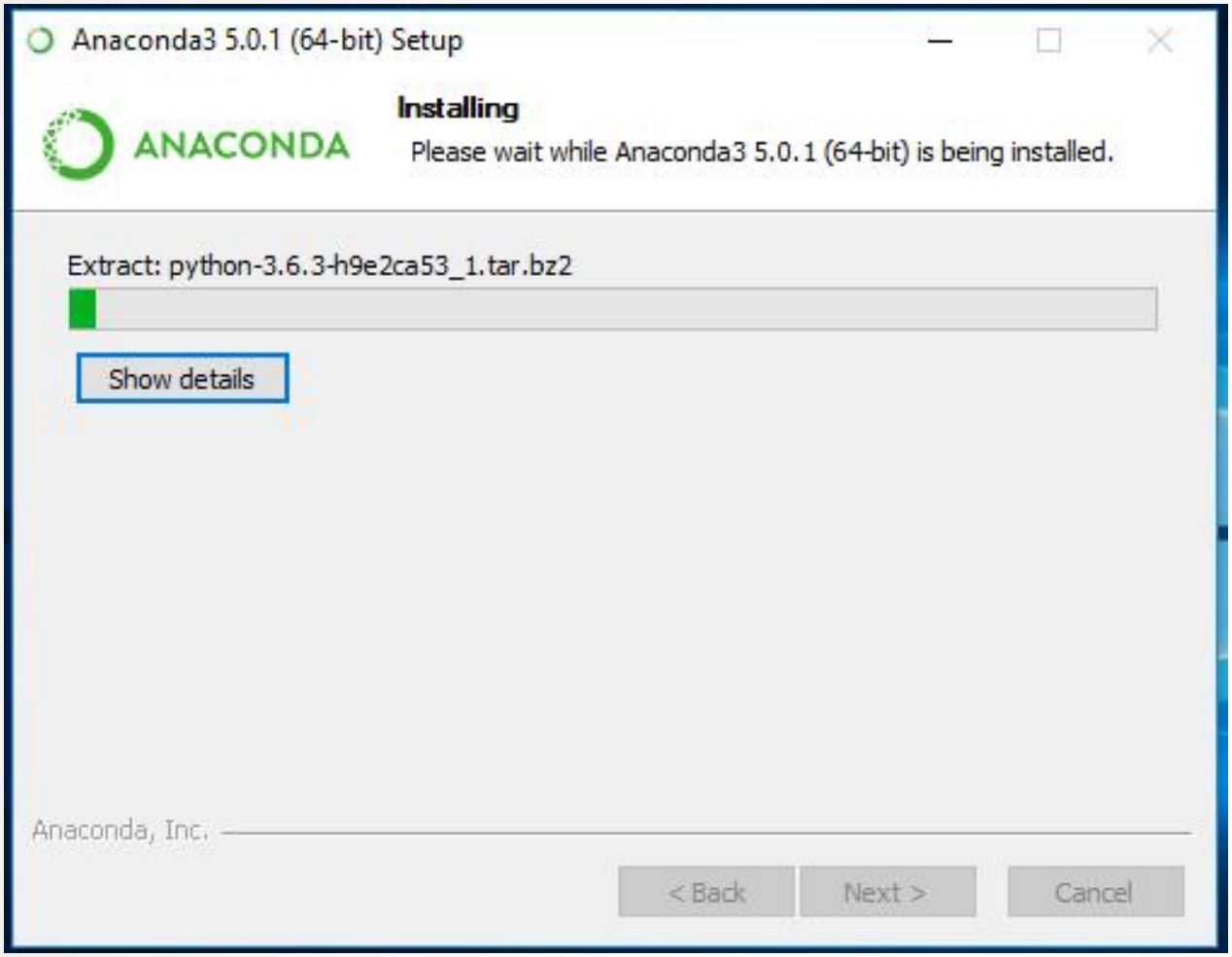
INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER



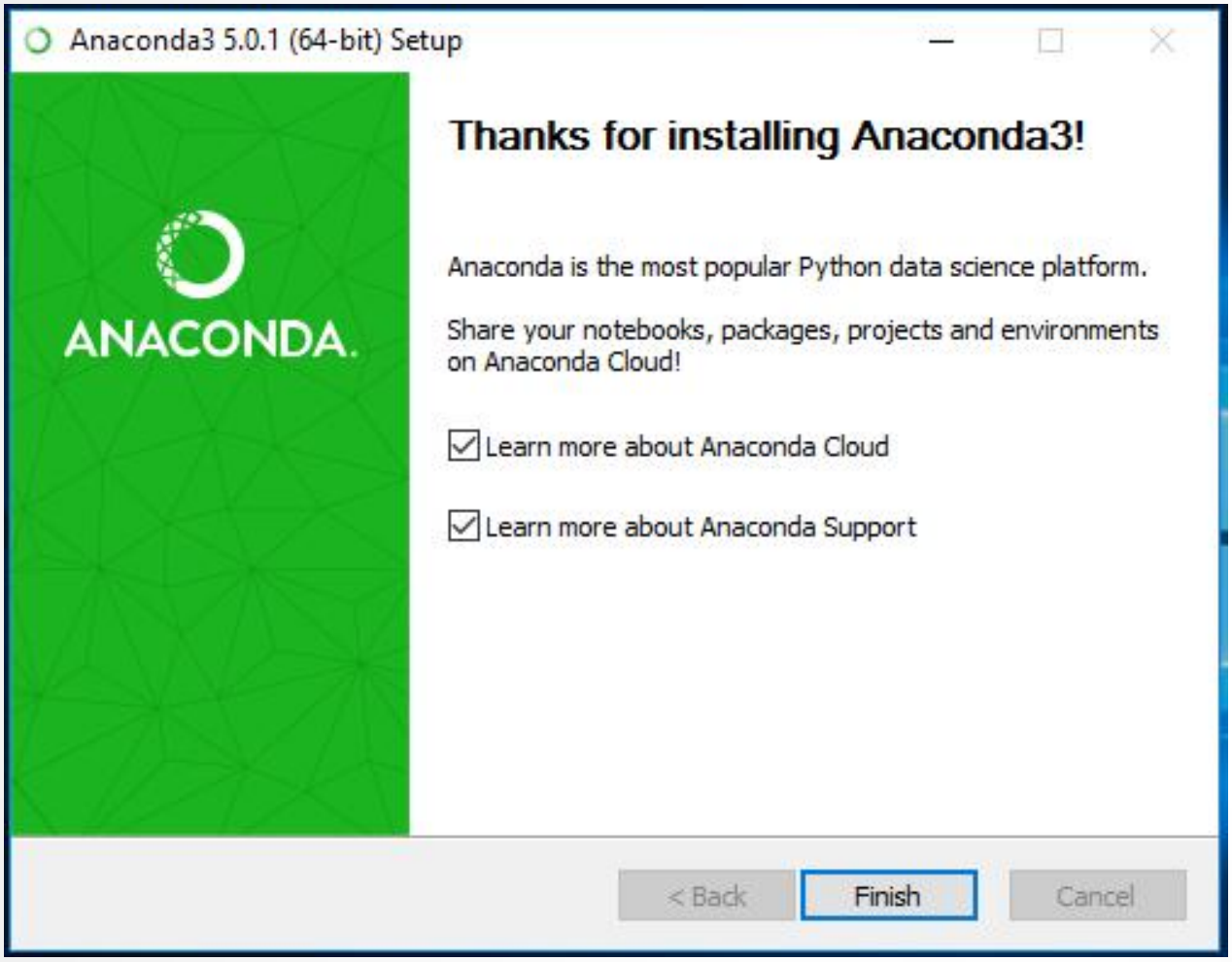
INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER



INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

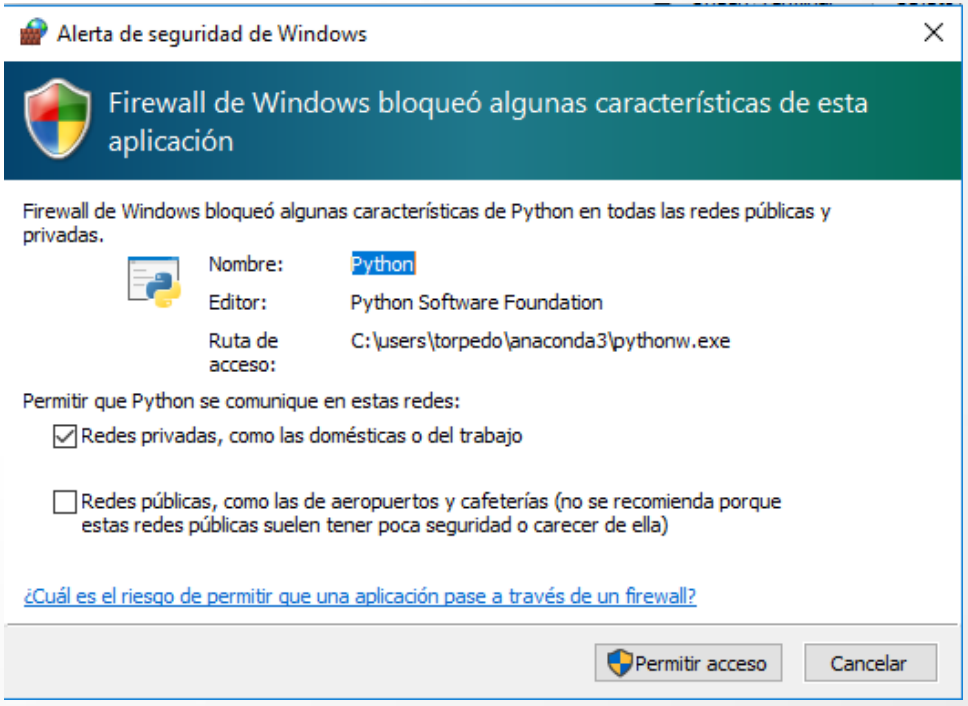


INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER



INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

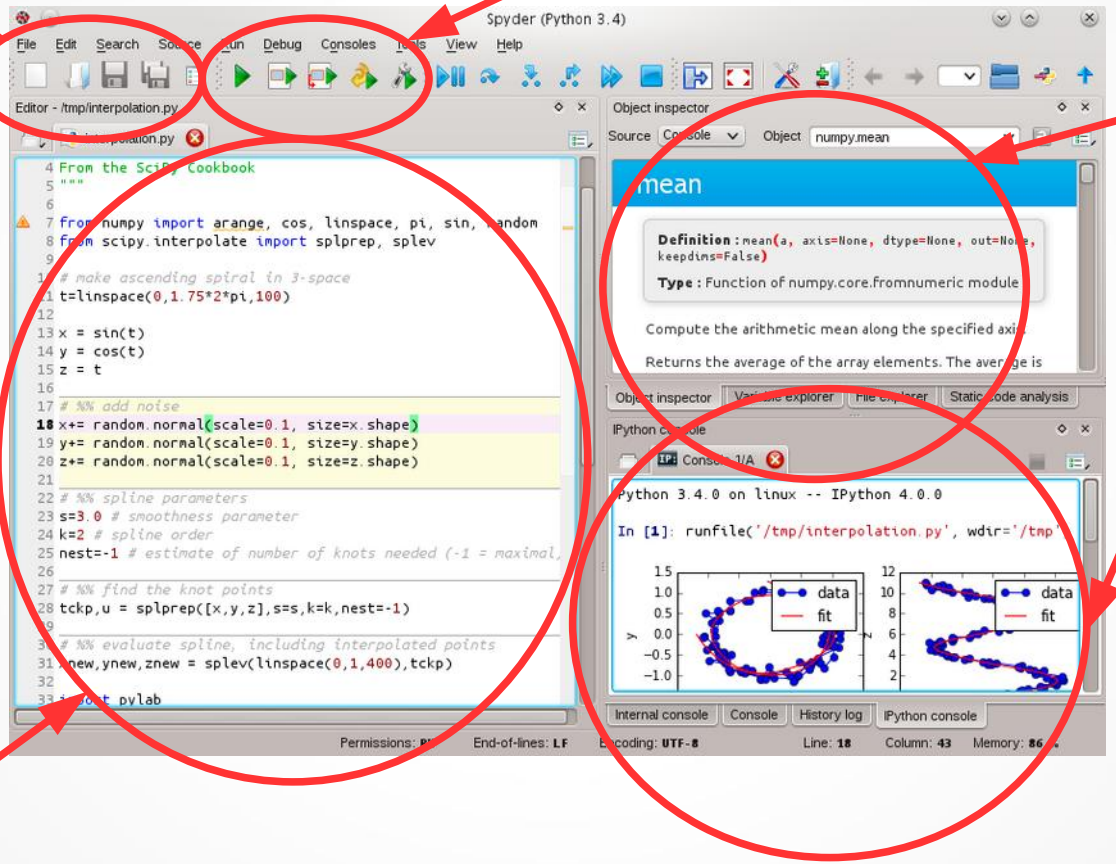
- Una vez instalado Anaconda, podemos encontrar spyder, en el menú de programas (buscar el icono spyder).
- Si al abrir el programa nos aparece un mensaje como el siguiente, pulsar en Permitir Acceso.



INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

Nuevo, Abrir, Guardar

Ejecución de programas



Inspector de objetos

Terminal de Python

Editor de programas

INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

•Podemos acceder a la ayuda y a un tutorial de Spyder (en inglés), a través de:

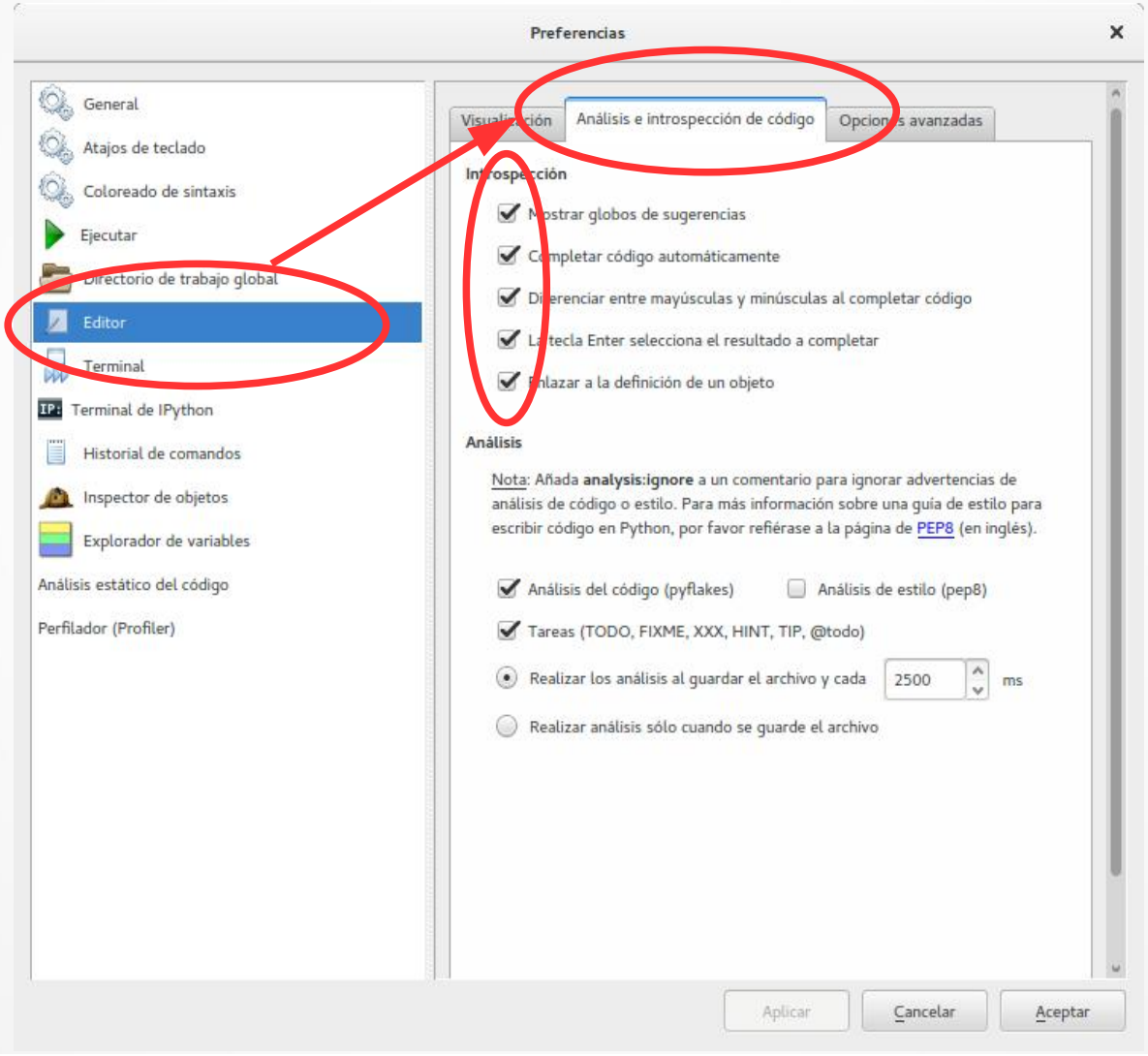
Ayuda – Documentación Spyder (F1)

Ayuda – Tutorial de Spyder.

•Un opción muy interesante es activar la ayuda en el **Inspector de Objetos, 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.

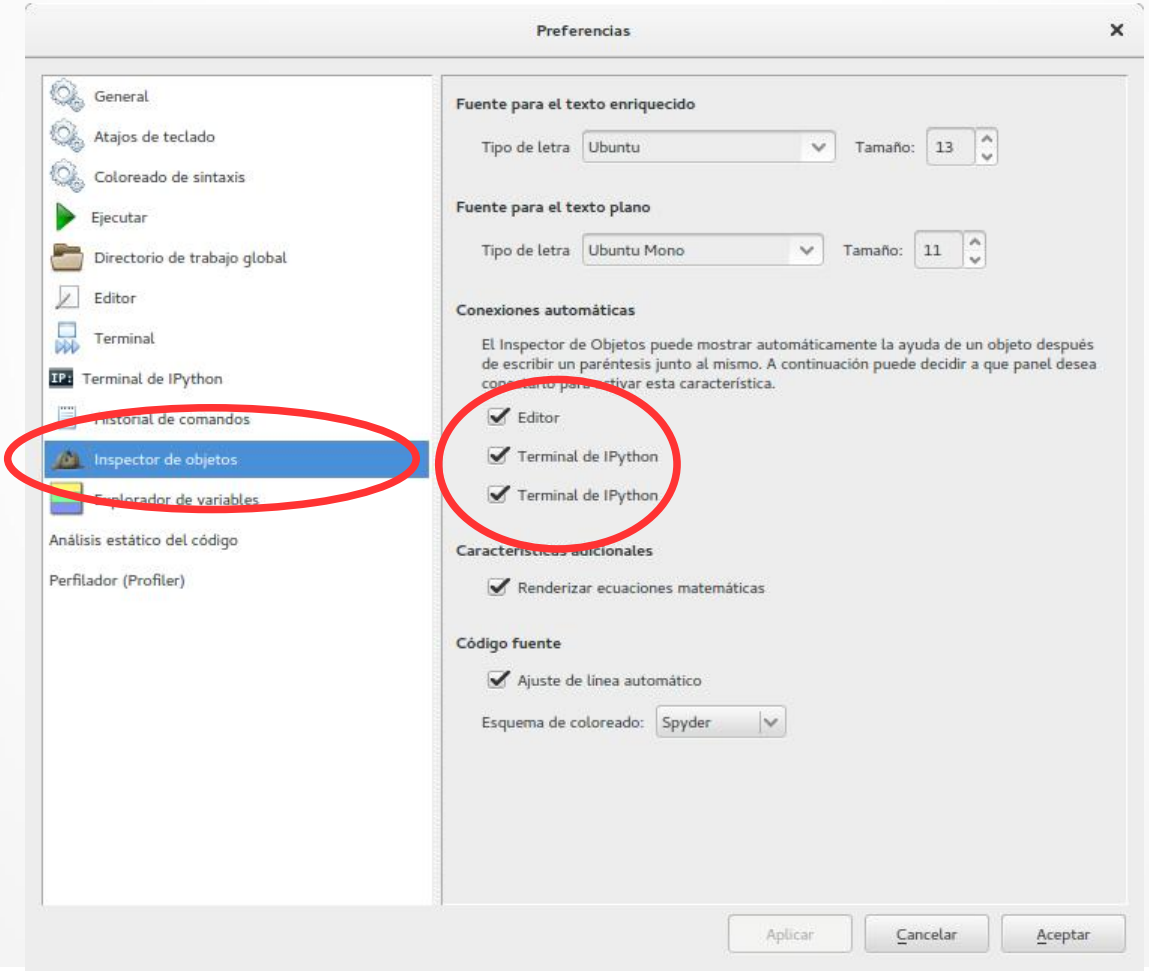
INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

→ Herramientas – Preferencias – Editor – Análisis e introspección de código – Introspección (Activar todas las opciones).



INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

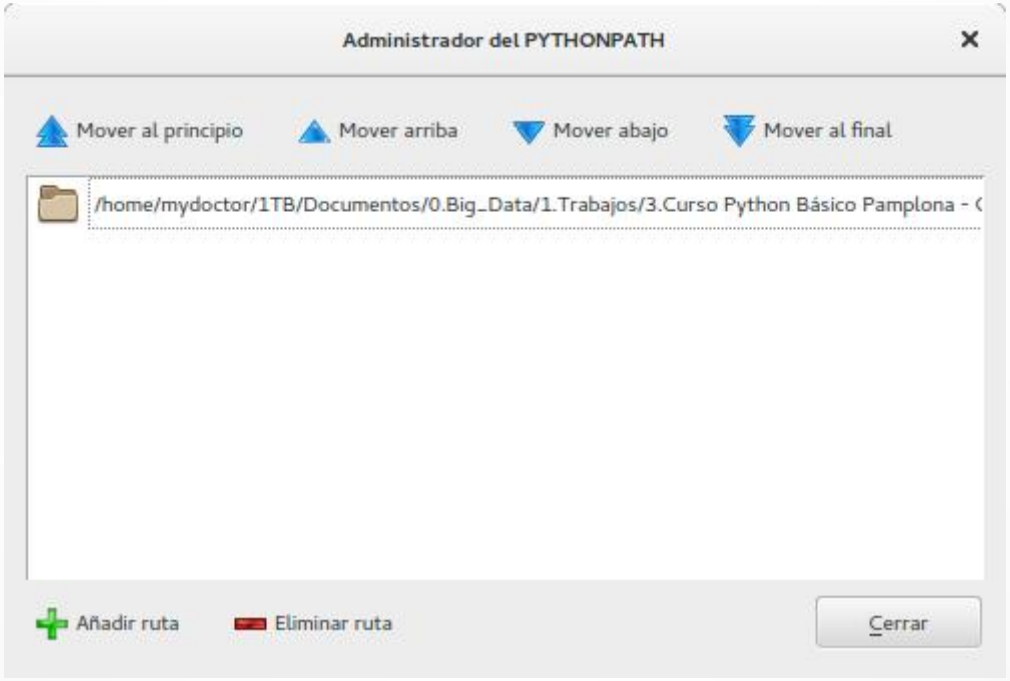
→ Herramientas – Preferencias – Inspector de objetos – Conexiones automáticas (Activar todas las opciones (Editor, Terminal de Ipython, Terminal de Ipython)



INTRODUCCIÓN – INSTALACIÓN PYTHON Y COMP. PRINCIPALES SPYDER

•Ruta por defecto de nuestros archivos. Permite configurar Python con las rutas donde ubicamos nuestros ficheros.

Herramientas – Administrador del PYTHONPATH



INTRODUCCIÓN – PAQUETES INCLUIDOS EN ANACONDA

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

INTRODUCCIÓN – PAQUETES INCLUIDOS EN ANACONDA

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

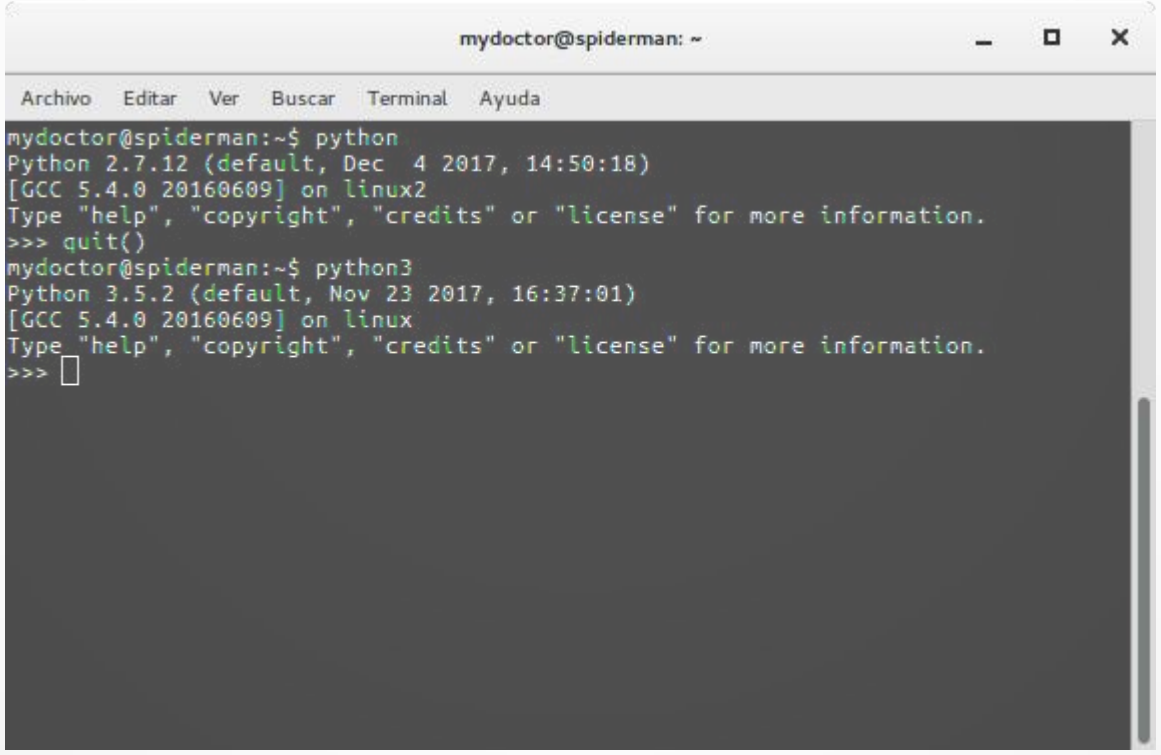
INTRODUCCIÓN – PAQUETES INCLUIDOS EN ANACONDA

- 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`**

INTRODUCCIÓN – CONSOLA/TERMINAL

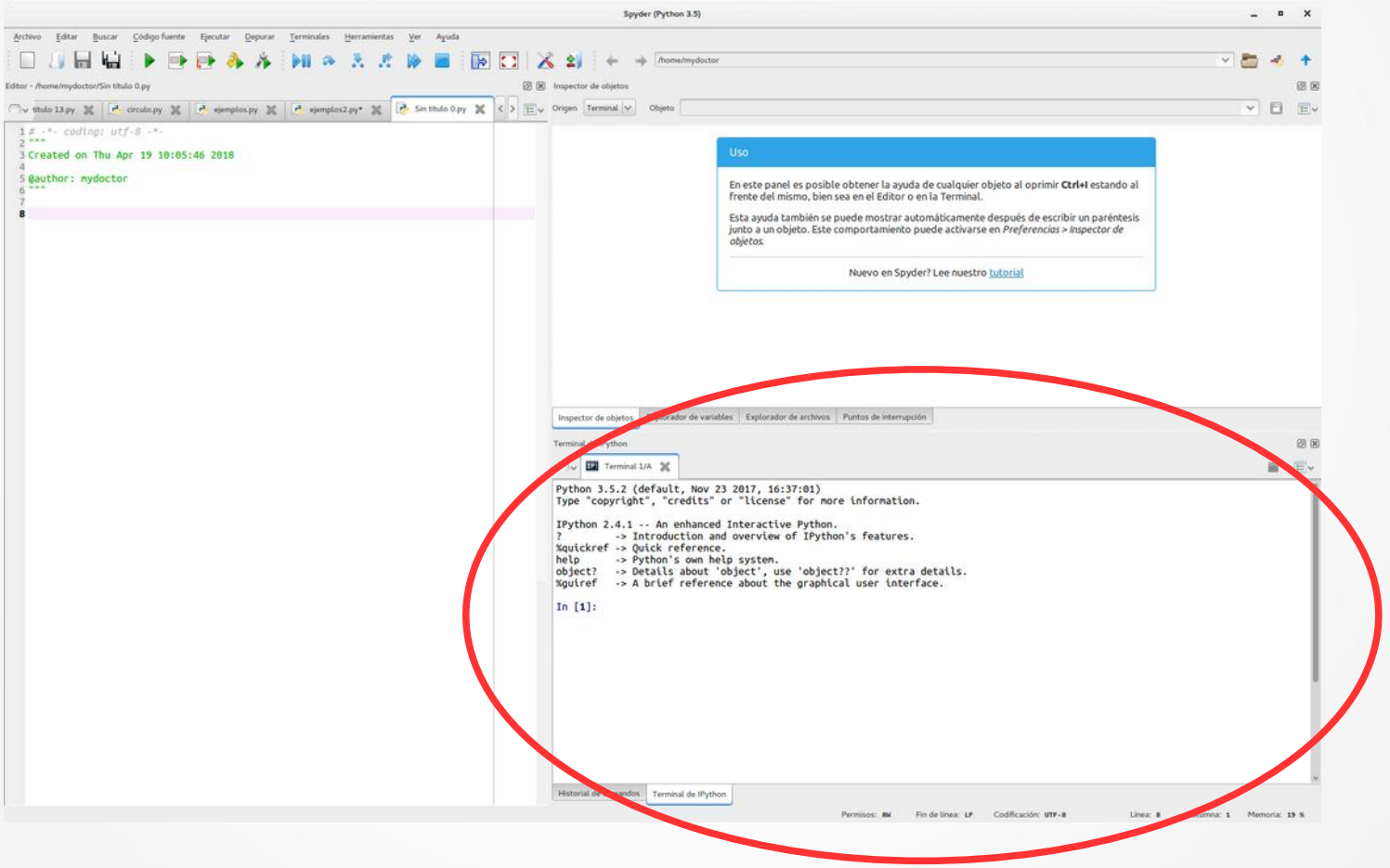
•Con Python instalado a través de Anaconda, podemos hacer uso de la consola de 2 maneras:

1)Abrir un Terminal de Anaconda, escribir **python** y pulsar **`Enter`**.



INTRODUCCIÓN – CONSOLA/TERMINAL

2)Abrir Spyder. La consola está situada en la esquina inferior derecha del IDE.



INTRODUCCIÓN – 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().`**

INTRODUCCIÓN – 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.

INTRODUCCIÓN – ELEMENTOS DE UN PROGRAMA PYTHON

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

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

INTRODUCCIÓN – ELEMENTOS DE UN PROGRAMA PYTHON

- El lenguaje Python está formado por elementos de diferentes tipos:
  - 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

INTRODUCCIÓN – 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.*

INTRODUCCIÓN – 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.

INTRODUCCIÓN – TIPOS DE VARIABLES

•Dentro de los **datos numéricos**, Python admite los siguientes tipos:

1)Enteros (int)
2)Coma flotante (float)
3)Complejos (complex)

Entero (int)	Coma flotante (float)	Complejo (complex)
10	0.0	3.14j
-900	11.10	23j
0x20	-99.9	2.34e-5j
-0410	1.2+e10	3e+88j

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Cadenas de texto:**
  - Una cadena de texto es una 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?+&”

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Listas:**
  - Las listas son estructuras de datos muy flexibles. Son conjuntos **ordenados** de elementos (números, cadenas, listas, …).
  - Pueden contener elementos no homogéneos (p.ej. números y cadenas)
  - Las listas son **MUTABLES** e iterables.

INTRODUCCIÓN – 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]]]
```

INTRODUCCIÓN – 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
```

INTRODUCCIÓN – 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.*

INTRODUCCIÓN – 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*

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Diccionarios:**
  - 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).

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Diccionarios:**

```
Ej: {'Piloto 1':"Fernando Alonso", 'Piloto 2':'Carlos Sainz', 'Piloto 3':'Kimi Raikkonen'}
```

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

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Diccionarios:**
  - 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.

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Diccionarios:**

LISTAS

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

DICCIONARIOS

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

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Diccionarios:**
  - 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.

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Diccionarios 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.

INTRODUCCIÓN – TIPOS DE VARIABLES

- **Conversión de datos a diferentes formatos:**

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

INTRODUCCIÓN – 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)
```

INTRODUCCIÓN – 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)
```

TABLA DE CONTENIDOS

2)Operadores básicos

i) Aritméticos
ii) De comparación
iii)De asignación
iv)Lógicos
v) De pertenencia
vi)De identidad
vii)Comandos y operadores básicos
viii)Aritméticos

a) De comparación
b) De pertenencia
c) De identidad
d) Operando con cadenas de texto
e) Operando con listas
f) Operando con tuplas
g) Operando con diccionarios
h) Print
i) Input
j) range()
k) Operadores comunes en cadenas, tuplas, rangos y listas

OPERADORES BÁSICOS - ARITMÉTICOS

- **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`

OPERADORES BÁSICOS - DE COMPARACIÓN

- **Operadores básicos para comparar elementos:**

Operador	Descripción	Ejemplo
`==`	Devuelve True, si el valor de los 2 operadores es igual	`a == b`
`!=`	Devuelve True, si el valor de los 2 operadores NO es igual	`a != b`
`<>`	Igual a !=	`a <> b`
`>`	Devuelve True, si el operador izquierdo es mayor al derecho	`a > b`
`<`	Devuelve True, si el operador izquierdo es menor al derecho	`a < b`
`>=`	Devuelve True, si el operador izquierdo es mayor o igual al derecho	`a >= b`
`<=`	Devuelve True, si el operador izquierdo es menor o igula al derecho	`a <= b`

OPERADORES BÁSICOS - DE ASIGNACIÓN

- **Operadores básicos de asignación:**

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

OPERADORES BÁSICOS - LÓGICOS

- **Operadores básicos lógicos:**

Operador	Descripción	Ejemplo
`and`	Conjunción	`1 > 2 and 3 > 2 → False`
`or`	Disyunción	`1 > 2 or 3 > 2 → True`
`not`	Negación	`Not 3 → True cuando lo que se evalúa no es igual a 3`

OPERADORES BÁSICOS – DE PERTENENCIA

- **Operadores básicos de pertenencia:**

Operador	Descripción	Ejemplo
`in`	Devuelve True si encuentra una variable dentro de una secuencia de datos	`a in b`
`not in`	Devuelve True si NO encuentra una variable dentro de una secuencia de datos	`a not in b`

OPERADORES BÁSICOS – DE IDENTIDAD

- **Operadores básicos de identidad:**

Operador	Descripción	Ejemplo
`is`	Devuelve 1 si ambos operadores a comparar son iguales	`a is b`
`is not`	Devuelve 0 si ambos operadores a comparar son diferentes	`a is not b`

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.

```
frase [0] → H
frase [5] → M
frase [-1] → o

frase [:] → ¿?
```

```
frase[:6] = Hola M
frase[2:7] = la Mu
   frase[2:] = la Mundo
```

COMANDOS Y OPERADORES BÁSICOS

- **Operando con cadenas de texto:**

→ Multiplicación con cadenas de texto:

```
saludo = “Hola”
3 * saludo = HolaHolaHola
```

→ Longitud de una cadena de texto, **len()**:

```
len (saludo)
4
```

→ Comparaciones (el resultado es un valor lógico):

```
saludo == 'Hola' → Devuelve True
'h' > 'f' → Devuelve True
'he' < 'ha' → Devuelve False
```

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.*

COMANDOS Y OPERADORES BÁSICOS

- **Operando con cadenas de texto:**

→ Crea un programa que compruebe si para una cadena de texto dada, existen las letras “i” o “u”.

```
cadena = "abcdefghijk"

for indice in range(len(cadena)):
   if cadena [indice] == "i" or cadena [indice] == "u":
      print ("La cadena contiene una i o una u")
```

COMANDOS Y OPERADORES BÁSICOS

- **Operando con cadenas de texto:**

El problema anterior también puede resolverse con:

```
for test in cadena:
 if test == "i" or test == "u":
  print ("La cadena contiene una i o una u")
```

➔ ¿Ves la diferencia entre ambos métodos?

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']
```

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.

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

COMANDOS Y OPERADORES BÁSICOS

- **Operando con listas:**

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

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
```

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).

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']
```

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

Método 2

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

Estamos iterando sobre los propios elementos de la lista directamente.

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.

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.

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).

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). Solo cambia lista3.

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.

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.

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.

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.

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]`

COMANDOS Y OPERADORES BÁSICOS

- **Operando con listas:**

  - Información ampliada sobre listas: https://docs.python.org/3/tutorial/datastructures.html

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",) (Necesario añadir la coma
                        al final de la tupla,
                        cuando sólo tenga un
                        elemento)
```

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)`

COMANDOS Y OPERADORES BÁSICOS

- **Operando con tuplas:**

Otra propiedad interesante, es que las tuplas forman un conjunto → `(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)
```

COMANDOS Y OPERADORES BÁSICOS

- **Operando con diccionarios:**

→ Añadir elementos:

```
diccionario = {'Piloto 1':"Fernando Alonso",
'Piloto 2':'Carlos Sainz', 'Piloto 3':'Kimi
Raikkonen'}
```

→ Añadir un nuevo piloto al diccionario,

```
diccionario ['Piloto 4'] = 'Sebastian Vettel'
```

```
diccionario → {'Piloto 1': 'Fernando Alonso',
               'Piloto 2': 'Carlos Sainz',
               'Piloto 3': 'Kimi Raikkonen'
               'Piloto 4': 'Sebastian Vettel'}
```

COMANDOS Y OPERADORES BÁSICOS

- **Operando con diccionarios:**

→ Comprobar si un elemento existe dentro del diccionario.

```
"Piloto 1" in diccionario
```

Devuelve **`True`**, si el nombre **`Piloto 1`** está en el diccionario.

→ Eliminar elementos de un diccionario.

```
del(diccionario['Piloto 1']) → no devuelve nada
diccionario → {'Piloto 2': 'Carlos Sainz',
 'Piloto 4': 'Sebastian Vettel',
 'Piloto 3': 'Kimi Raikkonen'}
```

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

COMANDOS Y OPERADORES BÁSICOS

- **Operando con diccionarios:**

→ Obtener las claves (keys) de un diccionario.

```
diccionario.keys() → dict_keys(['Piloto 2', 'Piloto 4', 'Piloto 3'])
```

→ Obtener los valores (values) de un diccionario.

```
diccionario.values() → dict_values(['Carlos Sainz', 'Sebastian Vettel', 'Kimi Raikkonen'])
```

→ Obtener el valor de una clave.

```
diccionario.get ('Piloto 4')
```

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")
```

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) separadas por un separador conocido (CSV, ...).

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

COMANDOS Y OPERADORES BÁSICOS

- **Print (imprimir información en pantalla):**

→ Imprimir cadenas de texto:

```
print (saludo, persona)
→ Hola Mundo

print (saludo + " " + sujeto)
→ Hola Mundo

print (saludo + sujeto)
→ HolaMundo
```

→ concatena cadenas

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)
```

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)
```

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.

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>
```

OPERADORES COMUNES EN CADENAS, TUPLAS, RANGOS Y LISTAS

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

- **`seq[i]`** → **`i`**^{avo} elemento de seq
- **`len(seq)`** → longitud de seq
- **`seq1 + seq2`** → concatencación (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

TABLA DE CONTENIDOS

3)Mutación, alias y clonación de listas
- i) Alias
- ii)Mutación
- iii)Clonación

4)Control de flujo – Condicionales

5)Iteraciones
- i) while
- ii)for
- iii)for anidado
- iv)break
- v)continue
- vi)Resumen
- vii)Ejemplo
- viii)“Guess and check”

MUTACIÓN, ALIAS Y CLONACIÓN DE LISTAS

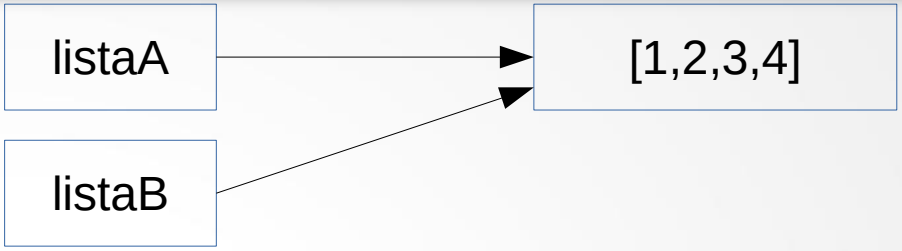
- ¿Qué sabemos de las listas hasta ahora?
  1. Son mutables (pueden cambiar).
  2. Se comportan de manera distinta a los tipos inmutables (tuplas).

     ---
  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.

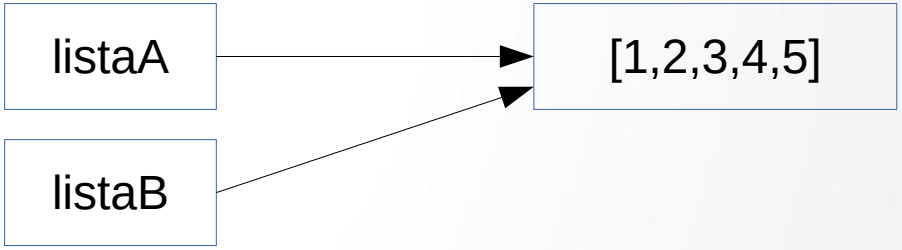
MUTACIÓN, ALIAS Y CLONACIÓN DE LISTAS - ALIAS

```
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.

MUTACIÓN, ALIAS Y CLONACIÓN DE LISTAS - ALIAS

•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.

MUTACIÓN, ALIAS Y CLONACIÓN DE LISTAS - ALIAS

•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]
```

MUTACIÓN, ALIAS Y CLONACIÓN DE LISTAS - MUTACIÓN

- 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]
```

MUTACIÓN, ALIAS Y CLONACIÓN DE LISTAS - MUTACIÓN

•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].

MUTACIÓN, ALIAS Y CLONACIÓN DE LISTAS - MUTACIÓN

- 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)

→ lista1: [3, 4]
→ lista2: [1, 2, 5, 6]
```

Cuando muta lista1, no lo hace lista1_copia

MUTACIÓN, ALIAS Y CLONACIÓN DE LISTAS - CLONACIÓN

- 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.

TABLA DE CONTENIDOS

3) Mutación, alias y clonación de listas
   i) Alias
   ii) Mutación
   iii) Clonación

4) Control de flujo – Condicionales

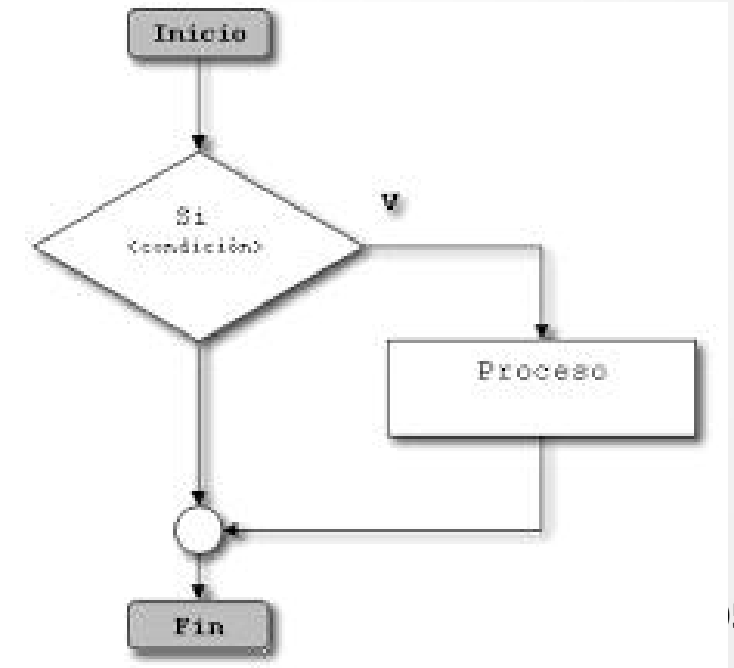
5) Iteraciones
   i) while
   ii) for
   iii) for anidado
   iv) break
   v) continue
   vi) Resumen
   vii) Ejemplo
   viii) “Guess and check”

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



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")
```

CONTROL DE FLUJO - CONDICIONALES

•El sangrado en el bloque de instrucciones, permite que el intérprete sepa que 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!")
```

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 2:
      sentencias a ejecutar
   else:
      sentencias a ejecutar
elif:
   sentencias a ejecutar
else: condición a evaluar 3:
   sentencias a ejecutar
      if condición a evaluar 4:
         sentencias a ejecutar
      else:
         break
sentencias a ejecutar
```

CONTROL DE FLUJO - CONDICIONALES

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

```
if condición a evaluar 1:
   sentencias a ejecutar
elif:
   sentencias a ejecutar
else:
   sentencias a ejecutar
```

TABLA DE CONTENIDOS

3) Mutación, alias y clonación de listas
   i) Alias
   ii) Mutación
   iii) Clonación

4) Control de flujo – Condicionales

5) Iteraciones
   i) while
   ii) for
   iii) for anidado
   iv) break
   v) continue
   vi) Resumen
   vii) Ejemplo
   viii) “Guess and check”

ITERACIONES

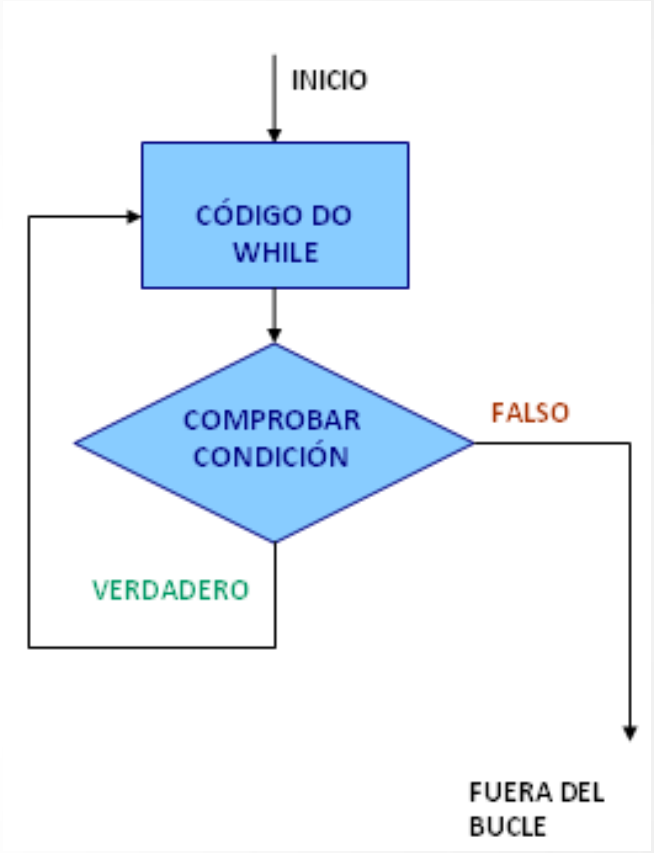
- Las iteraciones nos permiten ejecutar determinadas porciones de código de manera repetitiva.
- Son muy útiles porque permiten simplicar nuestro código y que éste sea más legible.
- Existen 2 tipos de iteraciones en Python:
    1)While
    2)For

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

ITERACIONES - WHILE

- **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).

ITERACIONES - WHILE

- **While**

Sintaxis:

while condición:
    cuerpo del bucle

Ejemplo:

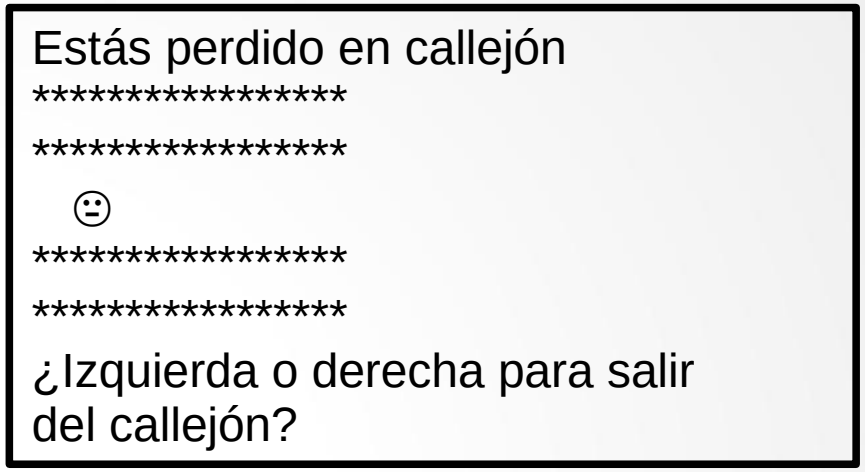
```
a = 0
while a < 10:
   print (a)
   a += 1
```

ITERACIONES - WHILE

- **While**

Ejemplo:

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




```
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")
```

ITERACIONES - WHILE

- **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.

ITERACIONES - WHILE

- **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#bltin-exceptions

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

ITERACIONES - WHILE

- **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...")
```

ITERACIONES - FOR

- **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.

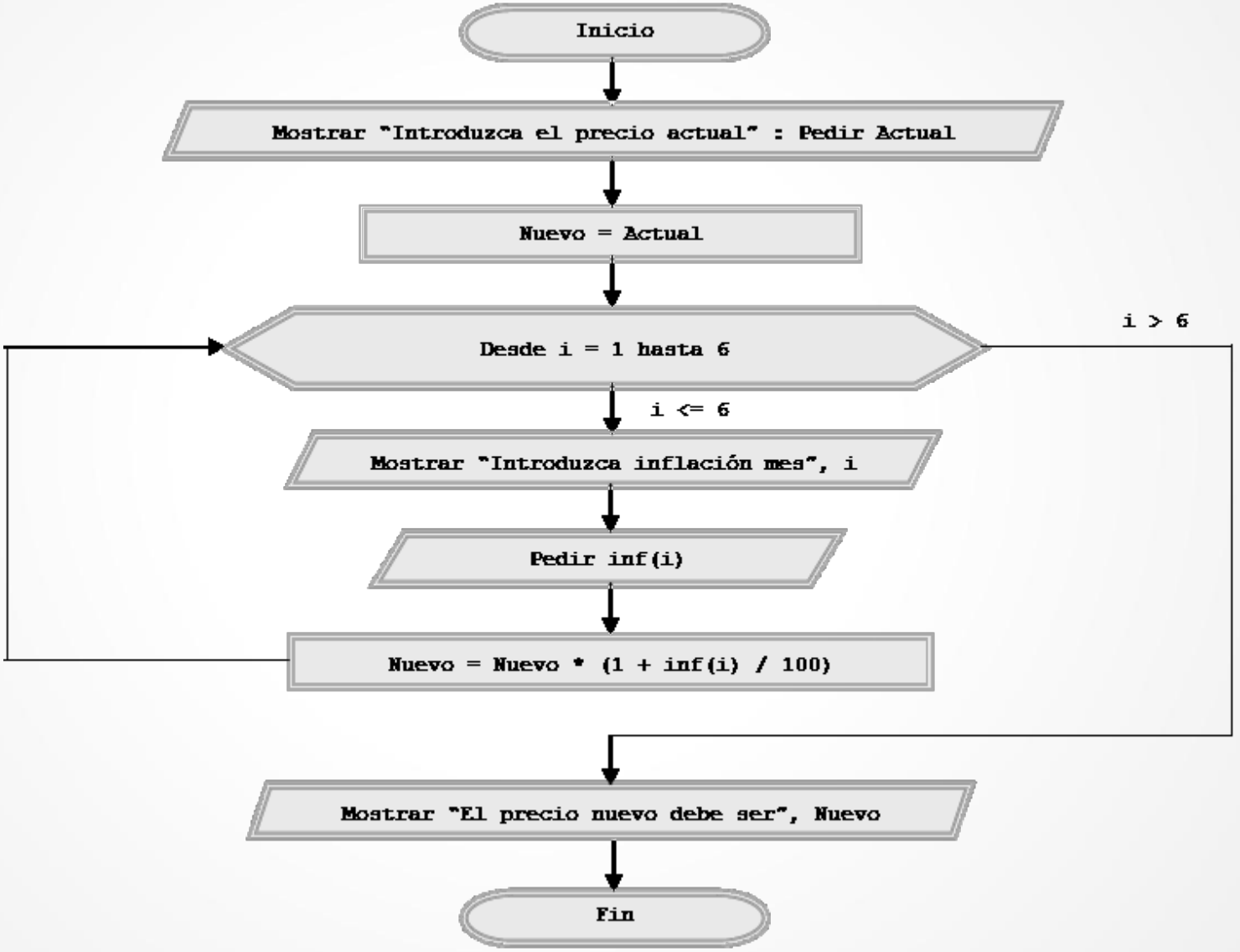
No es necesario definir variables de control, ya que la ejecución del bucle For, ES FINITA y definida.

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.

ITERACIONES - FOR

- **For**



ITERACIONES - FOR

- **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))
```

ITERACIONES – FOR ANIDADO

- **For**

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]
```

ITERACIONES – FOR ANIDADO

- **For**

Ejemplo:
Supongamos que queremos realizar la suma de los valores impares de los 10 primeros números.

```
b = 0
for a in range (1,10,2):
   print ("a:",a)
   b += a
   print ("b:",b)

a: 1
b: 1
a: 3
b: 4
a: 5
b: 9 ...
```

ITERACIONES – FOR ANIDADO

- **For**

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

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

Y los bucles For también pueden ser decrecientes:

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

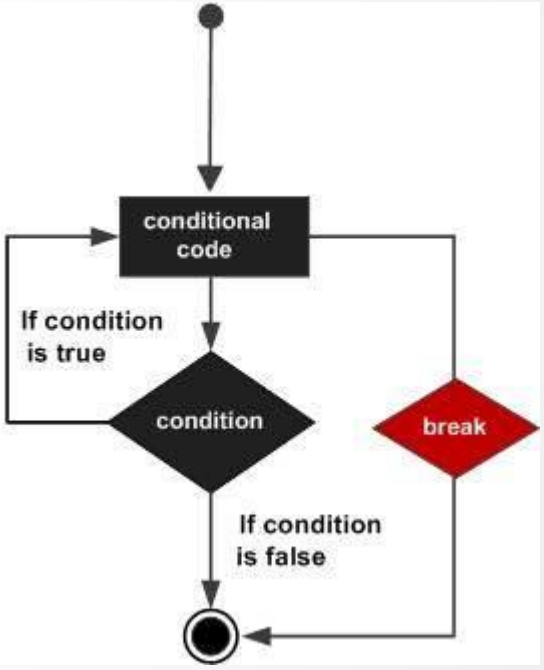
¡OJO! El último elemento del rango NO se evalua. En el último ejemplo, el bucle pararía en el caso inmediatamente anterior a 100 (105).

ITERACIONES – BREAK

- **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

ITERACIONES – BREAK

- **Break**

Ejemplo:

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

ITERACIONES – BREAK

- **Break**

En el caso de ejecutar Break dentro de loops anidados, 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!")
```

ITERACIONES – CONTINUE

- **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!")
```

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

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 **break**	Podemos cancelar la ejecución a través de **break**
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 **while**	Puede no ser posible reescribirse usando **for**

ITERACIONES – EJEMPLO

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

$2^2 = 2 + 2$ (dos, dos veces)

$3^2 = 3 + 3 + 3$ (tres, tres veces)

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

```
x = int (input("Introduce un valor entero: "))
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))
```

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

ITERACIONES – EJEMPLO

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

1. Definir la variable usada para la iteración, fuera del bucle (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.

ITERACIONES – EJEMPLO

- Reescribir el programa anterior para que ejecute el proceso a través de un FOR

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.

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 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

ITERACIONES – GUESS AND CHECK

•Crear un programa que nos 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.

ITERACIONES – GUESS AND CHECK

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

contador = 0

while contador**3 < x:
    contador = contador + 1
    print(contador)

if contador**3 != x:
    print(str(x) + ' no es un cubo perfecto')
else:
    print('El cubo de ' + str(x) + ' es ' +
str(contador))
```

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 lo 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).

ITERACIONES – GUESS AND CHECK

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

contador = 0

while contador**3 < abs(x):
    contador = contador + 1
    print(contador)

if contador**3 != abs(x):
    print(str(x) + ' no es un cubo perfecto')
else:
    if x < 0:
        contador = -contador
    print('El cubo de ' + str(x) + ' es ' +
str(contador))
```

ITERACIONES – GUESS AND CHECK

•Tenemos que tener presente, que DEBEMOS inicializar la variable a analizar (en el ejemplo: contadorº), si no lo hacemos el programa generará un error (del tipo NameError).

•Inicializar una variable ANTES de un bucle, nos permite controlar el rango de variación de dicha variable dentro del bucle.

•Permite arrancar el bucle, dentro de unos parámetros conocidos (¿Qué pasa si resultado = 100 al empezar el programa?).

•La variable a analizar, DEBE incrementarse (o reducirse) dentro del bucle. De otra manera acabaremos en un bucle infinito.

ITERACIONES – GUESS AND CHECK

- Este sistema NO es el más eficiente, ni puede usarse siempre, pero nos permite de una manera sencilla alcanzar objetivos.

ITERACIONES – GUESS AND CHECK

•Modificar el código del programa anterior para que calcule la raíz cúbica de un número entero (positivo o no) usando el método de “guess and check” a través de un **FOR.**

Crea una tabla como la de la diapositiva 130 para tener en cuenta todas las condiciones posibles.

ITERACIONES – GUESS AND CHECK

- Una modificación podría ser esta:

```
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?

ITERACIONES – GUESS AND CHECK

•IMPORTANTE. Cuando programemos, debemos tener en cuenta las condiciones y comportamiento que esperamos de él, y sobre todo EL QUE NO ESPERAMOS.

Pej: Si nuestro programa debe funcionar en el rango de datos 1 a 5, tendremos que considerar que pasaría si recibe datos fuera de ese rango.

TABLA DE CONTENIDOS

6)Funciones

i) Ámbito de las variables

ii)return vs. print

iii)Funciones como argumento

iv)Funciones como objetos

v)Valores por defecto

vi)Documentación de las funciones

7)Recursión

i) Recursión vs. Iteración

ii)Recursión para no numéricos.

FUNCIONES

•A medida que nuestros programas aumentan en complejidad y acumulan líneas de código, se hace cada vez menos fácil poder mantenerlo.
•Debemos tender a realizar código lo más limpio y simple posible.
•No es mejor un programa con mucho código, sino aquel que con el mismo código tiene mayor funcionalidad.

•Esto nos lleva a las **funciones.**

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.

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.

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.

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):

1)Hacer el bizcocho

2)Montar la nata

3)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).

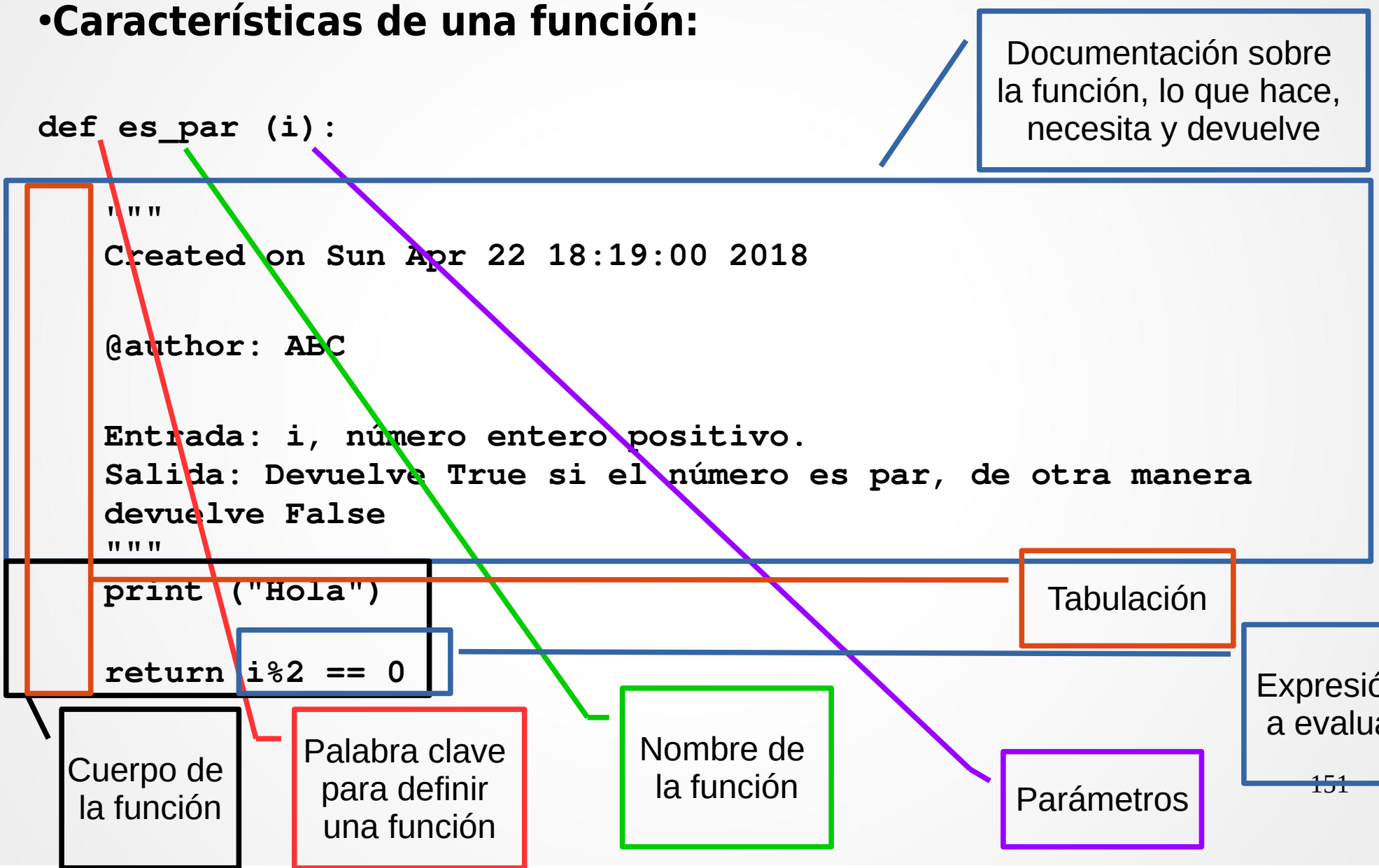
FUNCIONES

- **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.

FUNCIONES

- **Características de una función:**



FUNCIONES

- **Características de una función:**

  - Una vez creada la función, queda en memoria hasta que la llamamos.
  - En nuestro ejemplo, si escribimos en consola ***`es_par (2)`***, devolverá:
    `Hola`
    `True`

  - Una función, nos devolverá la expresión a evaluar en **`return`**

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.

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

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

•La función nos devuelve

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

•Si consultamos el valor de x, nos devuelve 3

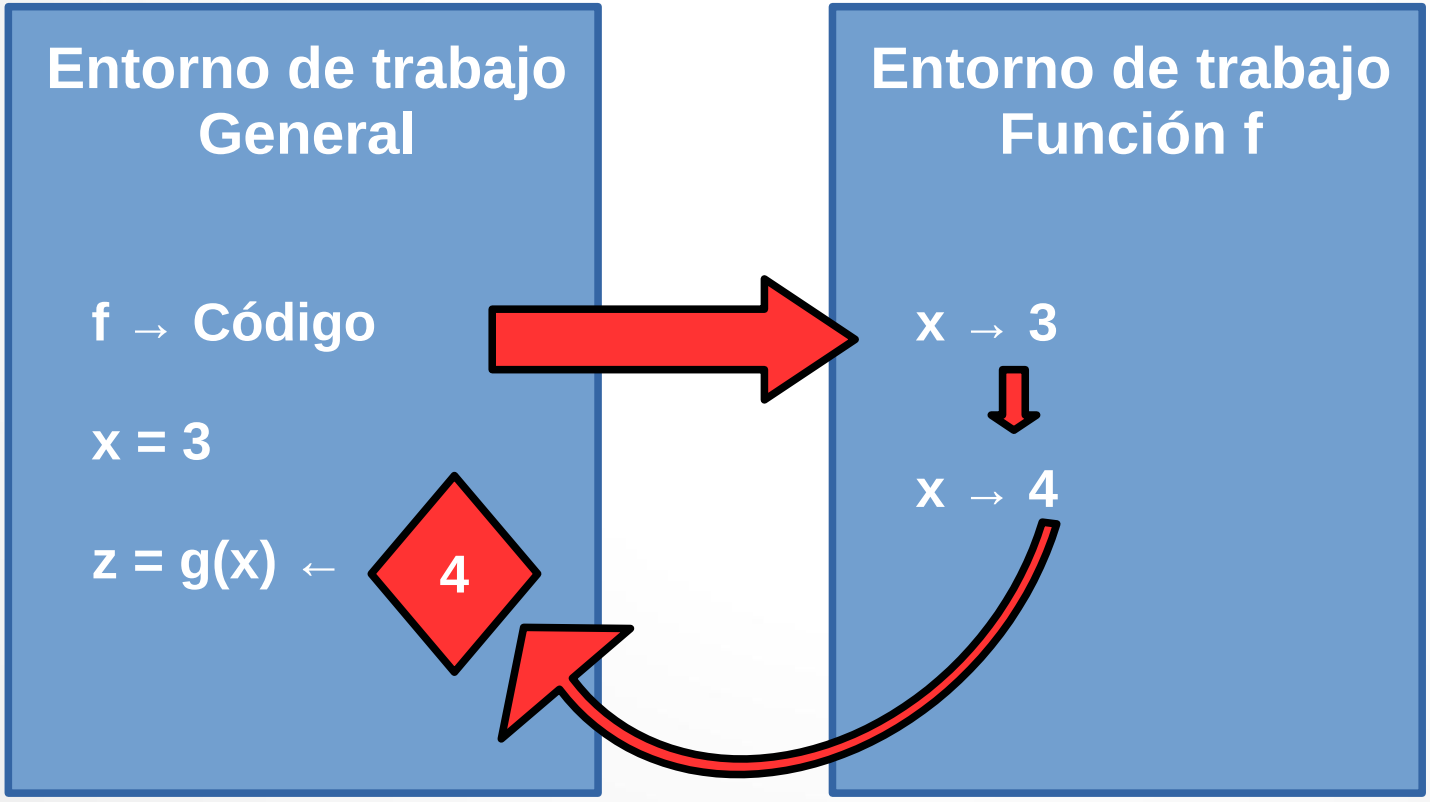
•¿Qué está pasando? ¿Porqué dentro de la función x=4 y fuera x=3, si parecen ser la misma variable?

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.

FUNCIONES – ÁMBITO DE LAS VARIABLES

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



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.

FUNCIONES – ÁMBITO DE LAS VARIABLES

def f(x): ***x = 1*** ***x += 1*** ***print (x)***	***def g(x):*** ***print (x)*** ***print (x + 1)***
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

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 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 una valor a la consola, pero no se almacena.

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	Devuelve
`print (func_a()) →`	`Dentro de la función a`
	`None`
`print (5 + func_b(2)→`	`Dentro de la función b`
	`7`
`print (func_c(func_a))→`	`Dentro de la función c`
	`Dentro de la función a`
	`None`

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.

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.

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.

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
```

FUNCIONES – COMO OBJETOS

- ¿Qué pasa si probamos la función previa como
  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.

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.

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.
•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.

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
```

TABLA DE CONTENIDOS

6) Funciones
   i) Ámbito de las variables
   ii) return vs. print
   iii) Funciones como argumento
   iv) Funciones como objetos
   v) Valores por defecto
   vi) Documentación de las funciones

7) Recursión
   i) Recursión vs. Iteración
   ii) Recursión para no numéricos.

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.

RECURSIÓN

•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).

•Ejemplo: Supongamos que queremos implementar una función que multiplica un número por si mismo n-veces. Vamos a comparar ambos métodos.

RECURSIÓN

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

```
a * b = a + a + a + … + a (b veces)

      = a + (a + a + … + a)   → (b-1 veces)
      = a + a * (b - 1)       → 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 → a * b = a

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

RECURSIÓN

ITERACIÓN

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

RECURSIÓN

```
def multi_recur (n,m):
    ''' n: número
        m: multiplicador
    '''

    if m == 1:
        return n
    else:
        return n +
multi_recur (n, m-1)
```

RECURSIÓN

•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:

1) Caso base donde n = 1

2) Simplificación del problema n! = n * (n-1)!

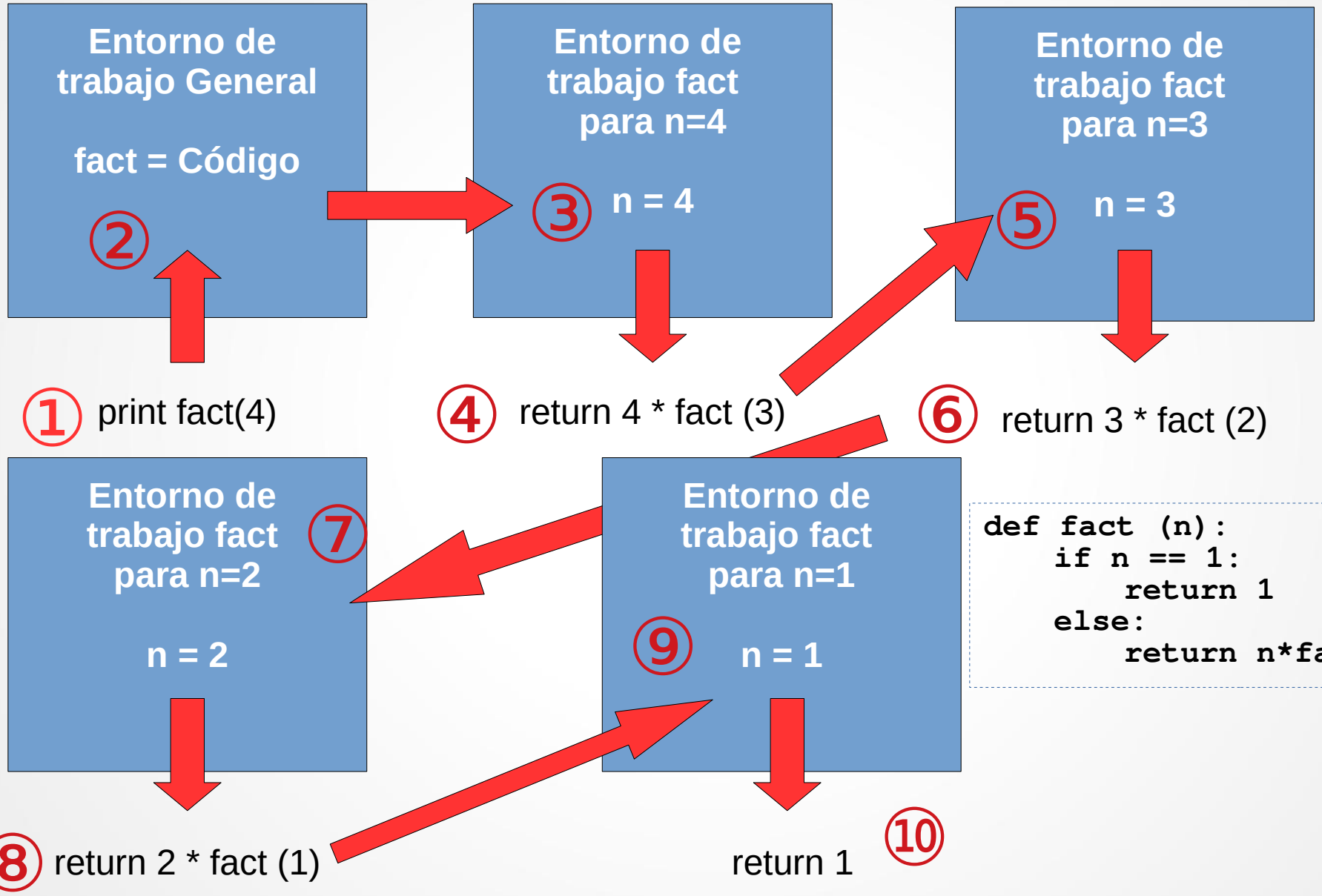
RECURSIÓN

- Esas 2 condiciones las podemos resumir en:

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

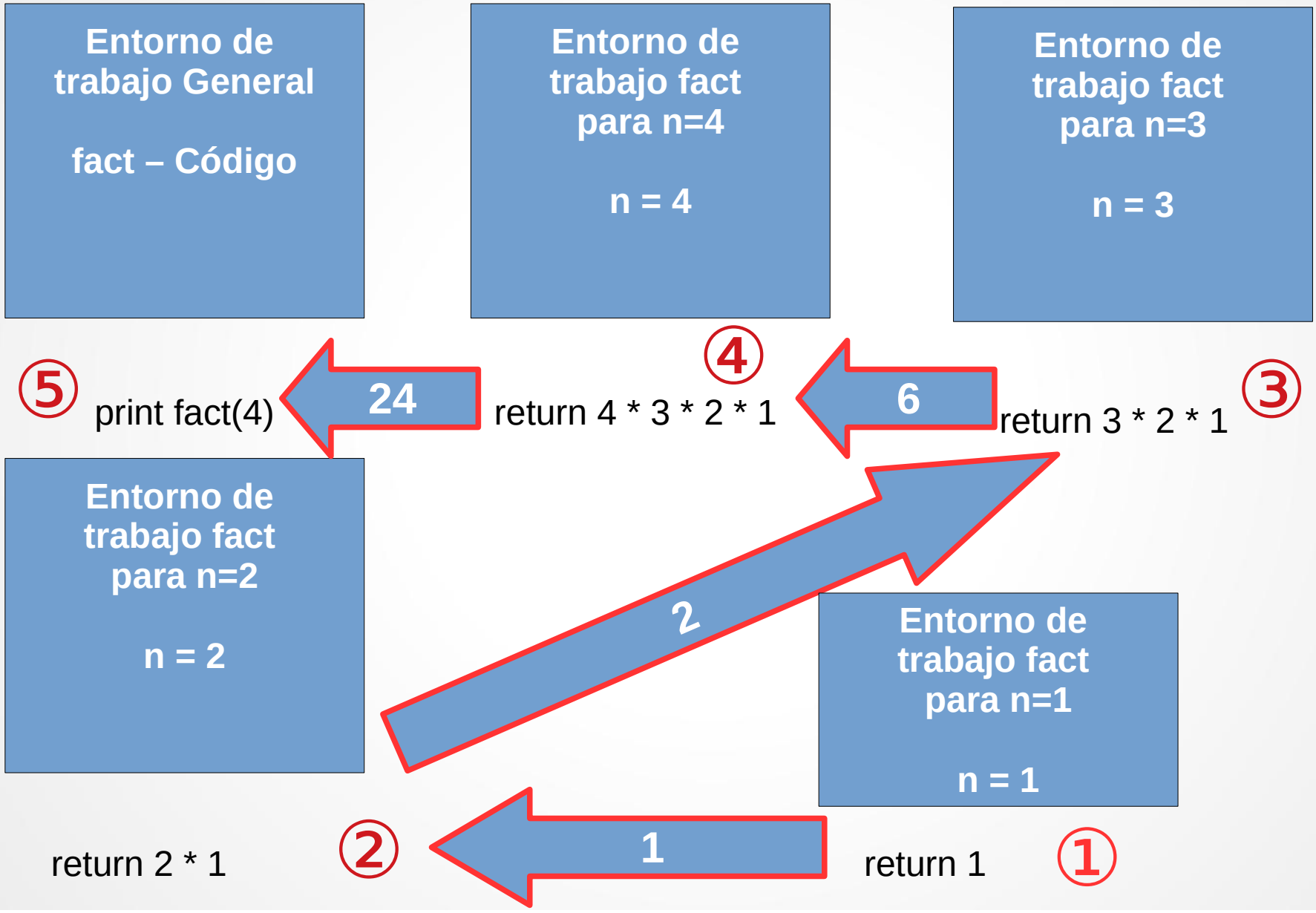
- ¿Cómo sabe el programa qué valor de n tomar en cada caso y cuándo terminar?. Entra en juego el ámbito de trabajo de las variables que vimos previamente.

RECURSIÓN




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

RECURSIÓN


Entorno de trabajo General
fact – Código
Entorno de trabajo fact para n=4
n = 4
Entorno de trabajo fact para n=3
n = 3
Entorno de trabajo fact para n=2
n = 2
Entorno de trabajo fact para n=1
n = 1
5
print fact(4)
24
4
return 4 * 3 * 2 * 1
6
3
return 3 * 2 * 1
2
2
return 2 * 1
1
1
return 1

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.

RECURSIÓN vs. ITERACIÓN

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)
```

RECURSIÓN PARA NO NUMÉRICOS

•Los algoritmos recursivos, puede utilizarse también con cadenas de texto.

•Básicamente (!) consiste en encontrar la manera de definir el caso base, y poder descomponer el problema en partes más pequeñas.

•Ejemplo: Programa que calcule si una cadena de texto es un palíndromo.

Consideraciones:

- Si la longitud de la cadena a analizar es 0 ó 1, la cadena será un palíndromo.
- Eliminamos de la cadena, signos de puntuación, exclamación, acentuación, espacios y convertimos todo a minúsculas.

RECURSIÓN PARA NO NUMÉRICOS

```
def esPalindromo(c):

    def aCaracter(c):
        c = c.lower()
        cadena = ''
        for a in c:
            if a in 'abcdefghijklmnopqrstuvwxyz':
                cadena = cadena + a
        return cadena

    def esPal(c):
        if len(c) <= 1:
            return True
        else:
            return c[0] == c[-1] and esPal(c[1:-1])

    return esPal(aCaracter(c))

→ ""
→ 'Dábale arroz a la zorra el abad'
→ 'Lavan esa base naval'
```

Las cadenas de texto, tienen la propiedad de poder convertirse a minúsculas y para ello usamos un método llamado lower

En loop, estamos elimi-nando cualquier carácter que no pertenezca a la cadena de referencia

Primero se genera una cadena válida y luego se disecciona por el principio y el final

RECURSIÓN PARA NO NUMÉRICOS

RETO: Prueba a crear un programa que haga lo mismo, pero de manera iterativa (FOR y/o WHILE) utilizando funciones o no.

Nota: Tendrás que tener en cuenta la longitud de la cadena a analizar.

TABLA DE CONTENIDOS

8)Trabajar con módulos y archivos
  i) Importar módulos
  ii)Trabajar con ficheros
    a)CSV
    b)Ficheros Excel

9)Introducción a la visualización de datos
  i) ¿Porqué visualizar los datos?
  ii)Estructura de un gráfico estadístico
  iii)Pylab

TRABAJANDO CON MÓDULOS Y ARCHIVOS

•En Python los archivos que generemos de denominan **módulos.**

•Los módulos tienen la extensión **.py.**

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

•Podemos crearnos 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**

IMPORTAR MÓDULOS

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

IMPORTAR MÓDULOS

•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
```

IMPORTAR MÓDULOS

•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`**).

IMPORTAR MÓDULOS

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
```

IMPORTAR MÓDULOS

•**Otros métodos 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.

TRABAJAR CON FICHEROS

•Necesitamos una manera de poder cargar y guardar el trabajo llevado a cabo.
•El objeto FILE, tiene una serie de métodos que nos permiten, abrir, leer, escribir, … ficheros.
•A través de la función OPEN, podemos cargar un archivo para después trabajar sobre él.

•**Abrir ficheros**

open ("nombre_archivo","w")

→ nombre_archivo = Archivo a abrir

→ w = Abrir el archivo en modo escritura

```
Ej: archivo = open ("prueba", "w")
```

TRABAJAR CON FICHEROS

•Ejemplo: Crear un fichero llamado nombres, que contenga los nombres de 3 personas.

```
archivo = open ("nombres","w")
for i in range(3):
    nombre = input("Introduce un nombre: ")
    archivo.write (nombre + "\n")

archivo.close()
```

r: solo lectura
w: lectura/escritura
a: añade al fichero
x: crea y escribe en un nuevo fichero

Añade líneas al archivo

Cierra el archivo creado y sobre escribe su contenido

TRABAJAR CON FICHEROS

•Si el archivo **nombres**, no existe, se crea través de **open**.

•**Cerrar ficheros**

Una vez finalizada la edición o adición de datos a un fichero, debemos cerrarlo para guardar los datos.

Para cerrar un fichero usaremos el método **close()**

```
archivo = open ("nombres","r")
for linea in archivo:
   print linea

archivo.close()
```

Itera por cada una de las líneas del fichero

Abrir archivo y cerrarlo automáticamente una vez leído:

```
with open ("nombre_archivo", "a")
```

TRABAJAR CON FICHEROS

- **Métodos más comunes del objeto File:**

  - **`read()`**: lee el contenido de un archivo.
  - **`readline()`**: lee una línea del archivo.
  - **`write()`**: escribe una cadena dentro del archivo.
  - **`close()`**: cierra un archivo

TRABAJAR CON FICHEROS - CSV

- Abrir ficheros en formato CSV:
  a) Importar el módulo **csv.**
  b) Importar el módulo **pandas.**

```
import csv
import pandas
archivo = pandas.read_csv ("archivo_csv")
print (archivo)
```

- Abrir ficheros sin cabeceras.

```
archivo = pandas.read_csv ("archivo_csv",
headers = None)
```

- Este módulo tiene muchísimas opciones más:
  https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

TRABAJAR CON FICHEROS - EXCEL's

•Podemos acceder a ficheros excel de varias maneras:

1) A través del módulo **pandas**

2) A través del módulo **xlrd**

•**pandas** viene instalado por defecto, **xlrd** no.

Instalación de xlrd

•La manera más sencilla es a través de una consola interactiva conda.

```
conda install -c anaconda xlrd
```

•Para entornos linux donde no existe esta consola, es necesario instalar el paquete **python-pip.**

•Posteriormente usar **`pip install xlrd --user`**

TRABAJAR CON FICHEROS - EXCEL's

- Abrir un archivo Excel a través de pandas.

```
import pandas as pd
fichero = pd.read_excel("nombre_archivo")
```

- Abrir un archivo Excel a través de xlrd.

```
from xlrd import open_workbook
fichero = open_workbook("nombre_fichero")
```

- Consultar pestañas existentes en un archivo.

```
fichero.sheet_names()
```

TRABAJAR CON FICHEROS - EXCEL’s

- Ayuda extendida sobre las funciones de este módulo:
  https://pandas.pydata.org/pandas-docs/version/0.20/generated/pandas.read_excel.html

TABLA DE CONTENIDOS

8) Trabajar con módulos y archivos
   i) Importar módulos
   ii) Trabajar con ficheros
      a) CSV
      b) Ficheros Excel

9) Introducción a la visualización de datos
   i) ¿Porqué visualizar los datos?
   ii) Estructura de un gráfico estadístico
   iii) Pylab

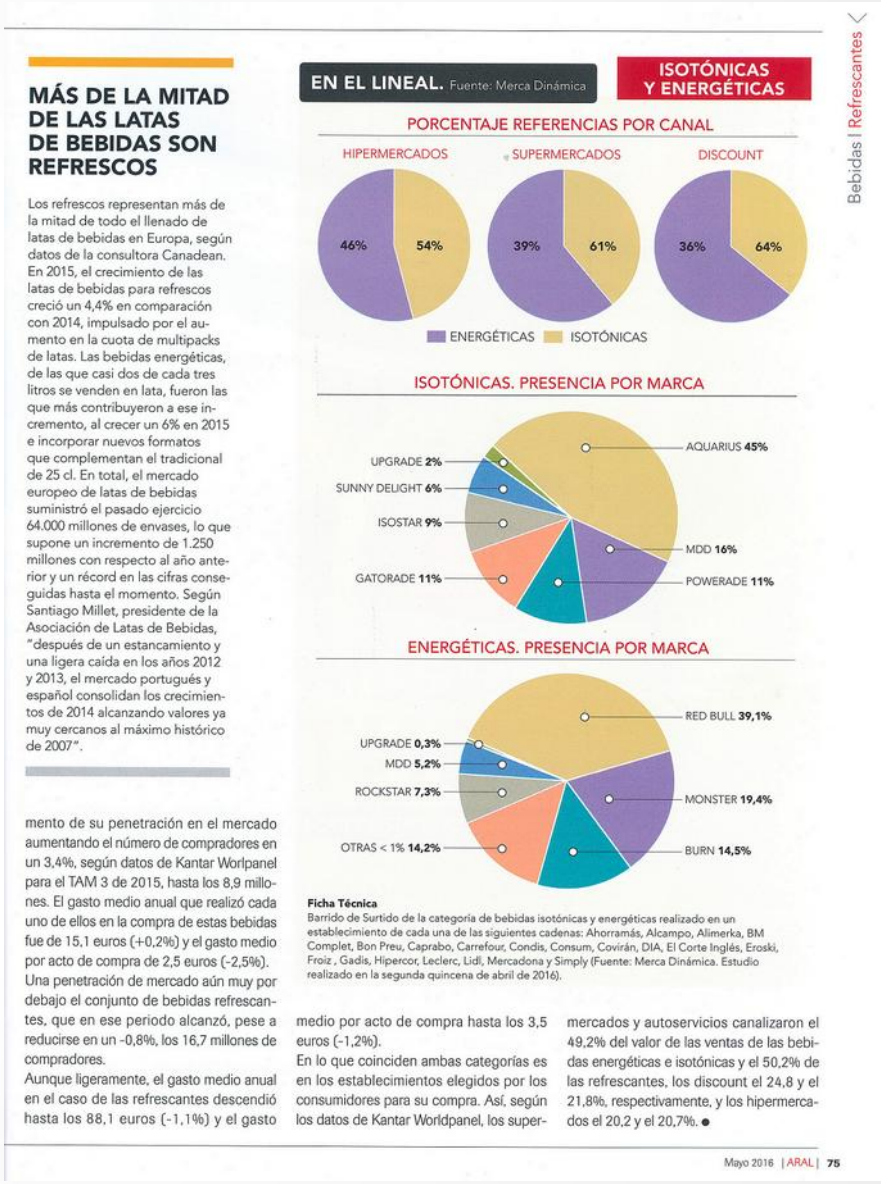
INTRODUCCIÓN A LA VISUALIZACIÓN DE DATOS

•La visualización de datos, nos permite visualizar la información que generemos con nuestros programas, de manera que pueda ser consumida y posibilite la toma de decisiones en los negocios.

•Explorar visualmente los datos, es una buena práctica para encontrar patrones, o comportamientos en las variables analizadas, que de otra manera sería complicado detectar.

INTRO VISUALIZACIÓN DE DATOS - ¿PORQUÉ VISUALIZAR LOS DATOS?

- Porque captan la atención.



Bebidas | Refrescantes

MÁS DE LA MITAD DE LAS LATAS DE BEBIDAS SON REFRESCOS

Los refrescos representan más de la mitad de todo el llenado de latas de bebidas en Europa, según datos de la consultora Canadean. En 2015, el crecimiento de las latas de bebidas para refrescos creció un 4,4% en comparación con 2014, impulsado por el aumento en la cuota de multipacks de latas. Las bebidas energéticas, de las que casi dos de cada tres litros se venden en lata, fueron las que más contribuyeron a ese incremento, al crecer un 6% en 2015 e incorporar nuevos formatos que complementan el tradicional de 25 cl. En total, el mercado europeo de latas de bebidas suministró el pasado ejercicio 64.000 millones de envases, lo que supone un incremento de 1.250 millones con respecto al año anterior y un récord en las cifras conseguidas hasta el momento. Según Santiago Millet, presidente de la Asociación de Latas de Bebidas, "después de un estancamiento y una ligera caída en los años 2012 y 2013, el mercado portugués y español consolidan los crecimientos de 2014 alcanzando valores ya muy cercanos al máximo histórico de 2007".

EN EL LINEAL. Fuente: Merca Dinámica

ISOTÓNICAS Y ENERGÉTICAS

PORCENTAJE REFERENCIAS POR CANAL

ISOTÓNICAS. PRESENCIA POR MARCA

ENERGÉTICAS. PRESENCIA POR MARCA

Ficha Técnica
Barrido de Surtido de la categoría de bebidas isotónicas y energéticas realizado en un establecimiento de cada una de las siguientes cadenas: Ahorramás, Alcampo, Alimerka, BM Complet, Bon Preu, Caprabo, Carrefour, Condis, Consum, Covirán, DIA, El Corte Inglés, Eroski, Froiz , Gadis, Hipercor, Leclerc, Lidl, Mercadona y Simply (Fuente: Merca Dinámica. Estudio realizado en la segunda quincena de abril de 2016).

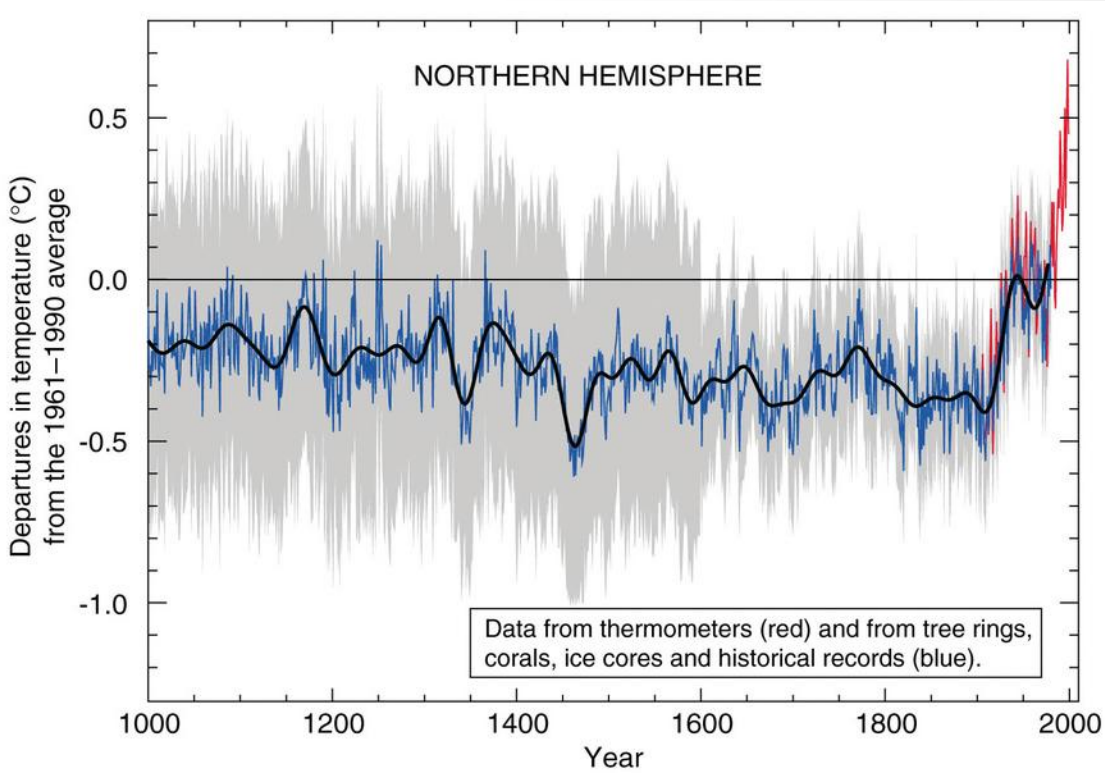
mento de su penetración en el mercado aumentando el número de compradores en un 3,4%, según datos de Kantar Worlpanel para el TAM 3 de 2015, hasta los 8,9 millones. El gasto medio anual que realizó cada uno de ellos en la compra de estas bebidas fue de 15,1 euros (+0,2%) y el gasto medio por acto de compra de 2,5 euros (-2,5%).
Una penetración de mercado aún muy por debajo el conjunto de bebidas refrescantes, que en ese periodo alcanzó, pese a reducirse en un -0,8%, los 16,7 millones de compradores.
Aunque ligeramente, el gasto medio anual en el caso de las refrescantes descendió hasta los 88,1 euros (-1,1%) y el gasto medio por acto de compra hasta los 3,5 euros (-1,2%).
En lo que coinciden ambas categorías es en los establecimientos elegidos por los consumidores para su compra. Así, según los datos de Kantar Worldpanel, los supermercados y autoservicios canalizaron el 49,2% del valor de las ventas de las bebidas energéticas e isotónicas y el 50,2% de las refrescantes, los discount el 24,8 y el 21,8%, respectivamente, y los hipermercados el 20,2 y el 20,7%. ●

INTRO VISUALIZACIÓN DE DATOS - ¿PORQUÉ VISUALIZAR LOS DATOS?

- Pueden ser memorables.

	A	B	C	D	E	F	G
1	YEAR	TEMP	YEAR	1 SIGMA	2 SIGMA		
2	1000	0.0659	1000	0.240346	0.480693	0.206137	0.123588
3	1001	-0.1241	1001	0.240347	0.480694	0.206137	0.123589
4	1002	-0.1208	1002	0.240346	0.480692	0.206136	0.123588
5	1003	-0.1801	1003	0.240347	0.480694	0.206137	0.123589
6	1004	-0.0711	1004	0.240347	0.480693	0.206137	0.123588
7	1005	-0.1334	1005	0.240346	0.480692	0.206136	0.123588
8	1006	-0.0644	1006	0.240346	0.480693	0.206137	0.123588
9	1007	0.0042	1007	0.240347	0.480693	0.206137	0.123588
10	1008	-0.1288	1008	0.240347	0.480693	0.206137	0.123588
11	1009	-0.0296	1009	0.240347	0.480693	0.206137	0.123588
12	1010	0.1187	1010	0.240347	0.480694	0.206137	0.123589
13	1011	-0.1252	1011	0.240346	0.480692	0.206136	0.123588
14	1012	-0.1634	1012	0.240347	0.480694	0.206137	0.123588
15	1013	-0.0791	1013	0.240347	0.480693	0.206137	0.123588
16	1014	-0.1120	1014	0.240347	0.480693	0.206137	0.123588
17	1015	-0.1146	1015	0.240346	0.480692	0.206136	0.123588
18	1016	-0.1206	1016	0.240346	0.480692	0.206136	0.123588
19	1017	-0.0815	1017	0.240347	0.480693	0.206137	0.123588
20	1018	-0.2031	1018	0.240346	0.480693	0.206137	0.123588
21	1019	0.0305	1019	0.240347	0.480693	0.206137	0.123588
22	1020	0.1098	1020	0.240347	0.480694	0.206137	0.123589
23	1021	0.0244	1021	0.240347	0.480693	0.206137	0.123588
24	1022	-0.0743	1022	0.240347	0.480693	0.206137	0.123588
25	1023	-0.0323	1023	0.240347	0.480693	0.206137	0.123588
26	1024	-0.0434	1024	0.240346	0.480693	0.206137	0.123588

878	1876	-0.1891	1876	0.113228	0.226456	8.25297E-02	7.75207E-02
879	1877	-0.0140	1877	0.113228	0.226457	8.25299E-02	7.75209E-02
880	1878	-0.0873	1878	0.113228	0.226457	8.25298E-02	7.75209E-02
881	1879	-0.2959	1879	0.113229	0.226458	8.25302E-02	7.75212E-02
882	1880	-0.2368	1880	0.113229	0.226457	8.25300E-02	7.75210E-02
883	1881	-0.1977	1881	0.113229	0.226458	8.25302E-02	7.75212E-02
884	1882	-0.2036	1882	0.113229	0.226457	8.25300E-02	7.75210E-02
885	1883	-0.2489	1883	0.113228	0.226455	8.25293E-02	7.75204E-02
886	1884	-0.2125	1884	0.113229	0.226457	8.25301E-02	7.75211E-02
887	1885	-0.1896	1885	0.113228	0.226457	8.25299E-02	7.75210E-02
888	1886	-0.1084	1886	0.113228	0.226456	8.25298E-02	7.75208E-02
889	1887	-0.3265	1887	0.113228	0.226456	8.25296E-02	7.75206E-02
890	1888	-0.1694	1888	0.113228	0.226457	8.25298E-02	7.75209E-02
891	1889	-0.1339	1889	0.113228	0.226456	8.25298E-02	7.75208E-02
892	1890	-0.3107	1890	0.113229	0.226457	8.25301E-02	7.75211E-02
893	1891	-0.1754	1891	0.113229	0.226457	8.25300E-02	7.75210E-02
894	1892	-0.3186	1892	0.113228	0.226456	8.25295E-02	7.75205E-02
895	1893	-0.3236	1893	0.113228	0.226456	8.25297E-02	7.75207E-02
896	1894	-0.1970	1894	0.113228	0.226456	8.25295E-02	7.75205E-02



INTRO VISUALIZACIÓN DE DATOS - ¿PORQUÉ VISUALIZAR LOS DATOS?

- Es la forma de pensar de los números.

	Set A		Set B		Set C		Set D	
	X	Y	X	Y	X	Y	X	Y
0	10	8.04	10	9.14	10	7.46	8	6.58
1	8	6.95	8	8.14	8	6.77	8	5.76
2	13	7.58	13	8.74	13	12.74	8	7.71
3	9	8.81	9	8.77	9	7.11	8	8.84
4	11	8.33	11	9.26	11	7.81	8	8.47
5	14	9.96	14	8.10	14	8.84	8	7.04
6	6	7.24	6	6.13	6	6.08	8	5.25
7	4	4.26	4	3.10	4	5.39	19	12.50
8	12	10.84	12	9.13	12	8.15	8	5.56
9	7	4.82	7	7.26	7	6.42	8	7.91
10	5	5.68	5	4.74	5	5.73	8	6.89
mean	9.00	7.50	9.00	7.50	9.00	7.50	9.00	7.50
std	3.32	2.03	3.32	2.03	3.32	2.03	3.32	2.03
corr	0.82		0.82		0.82		0.82	
lin. reg.	$y = 3.00 + 0.500x$		$y = 3.00 + 0.500x$		$y = 3.00 + 0.500x$		$y = 3.00 + 0.500x$	

INTRO VISUALIZACIÓN DE DATOS - ¿PORQUÉ VISUALIZAR LOS DATOS?

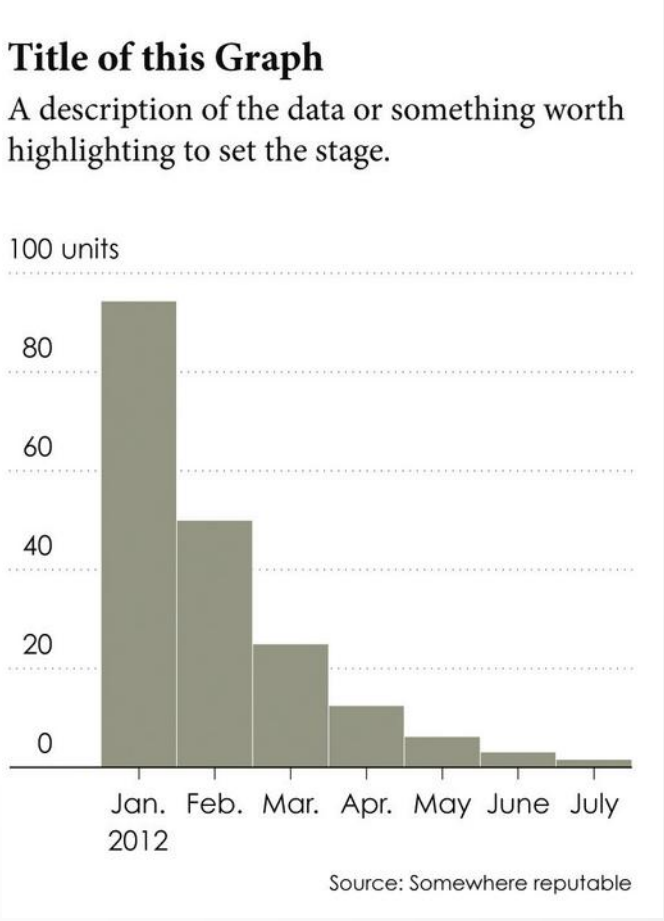
- Permiten ver los datos

85689726984689762689764358922659865986554897689269898
02462996874026557627986789045679232769285460986772098
90834579802790759047098279085790847729087590827908754
98709856749068975786259845690243790472190790709811450

85689726984689762689764  589226598659865548976892698980246299687402655762798678904567927692854609867720989345798027907590470982790857908477290875908279087549870985674906897578625984569024  790472190790709811450

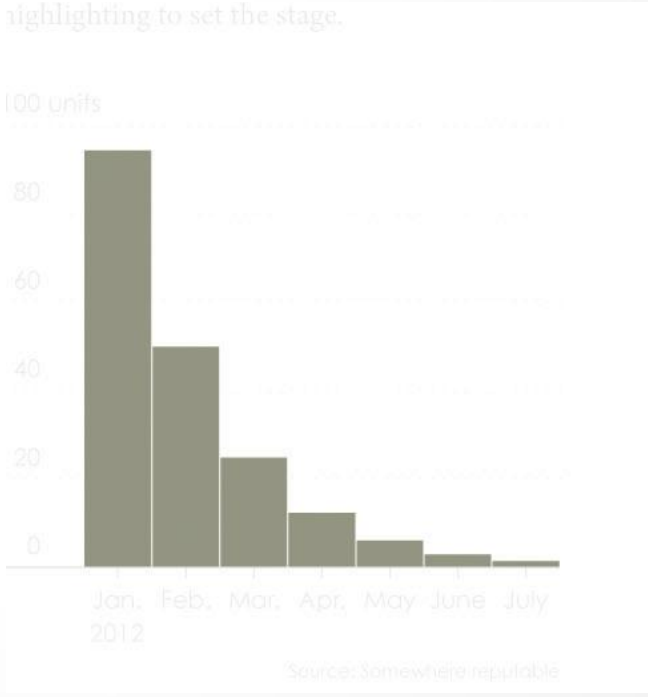
INTRO VISUALIZ. DE DATOS - ESTRUCTURA DE UN GRÁFICO ESTADÍSTICO

- **Título**: Descripción de los datos o algo que merezca la pena resaltar.



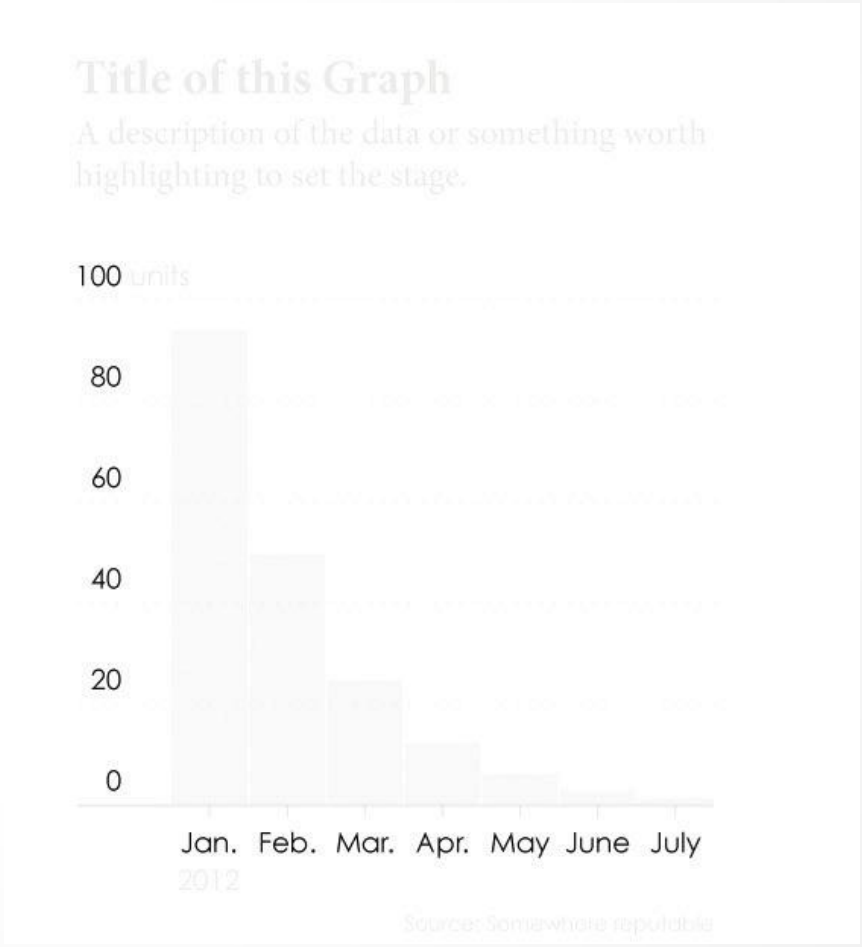
INTRO VISUALIZ. DE DATOS - ESTRUCTURA DE UN GRÁFICO ESTADÍSTICO

- **Estilo visual**: La visualización implica formatear los datos con formas colores y tamaños. Elegir un estilo u otro, dependerá de los datos y los objetivos.



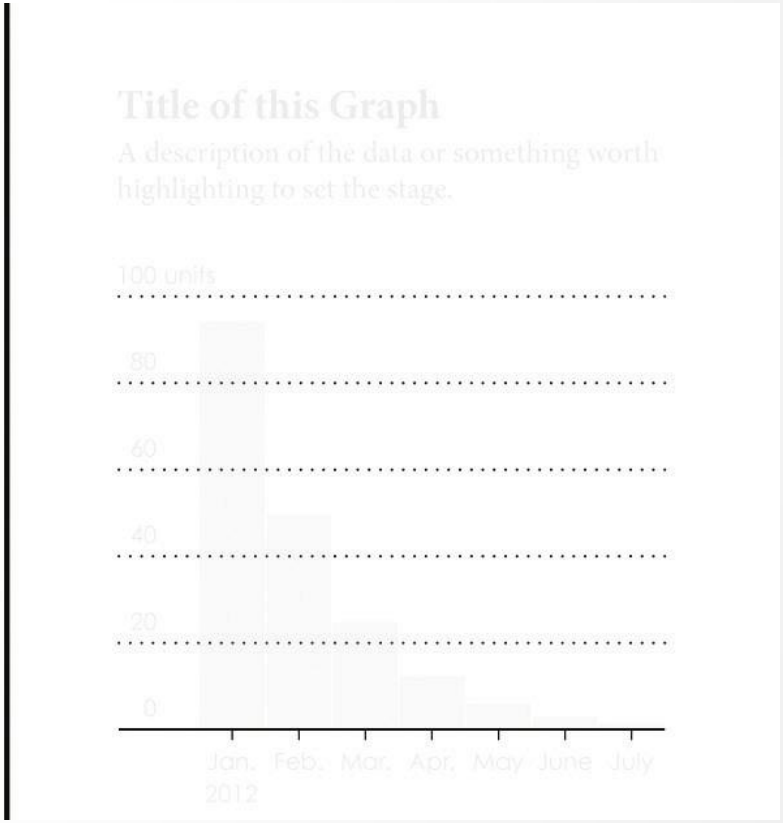
INTRO VISUALIZ. DE DATOS - ESTRUCTURA DE UN GRÁFICO ESTADÍSTICO

- **Escala**: La escala utilizada puede ayudar a mejorar la comprensión de los datos.



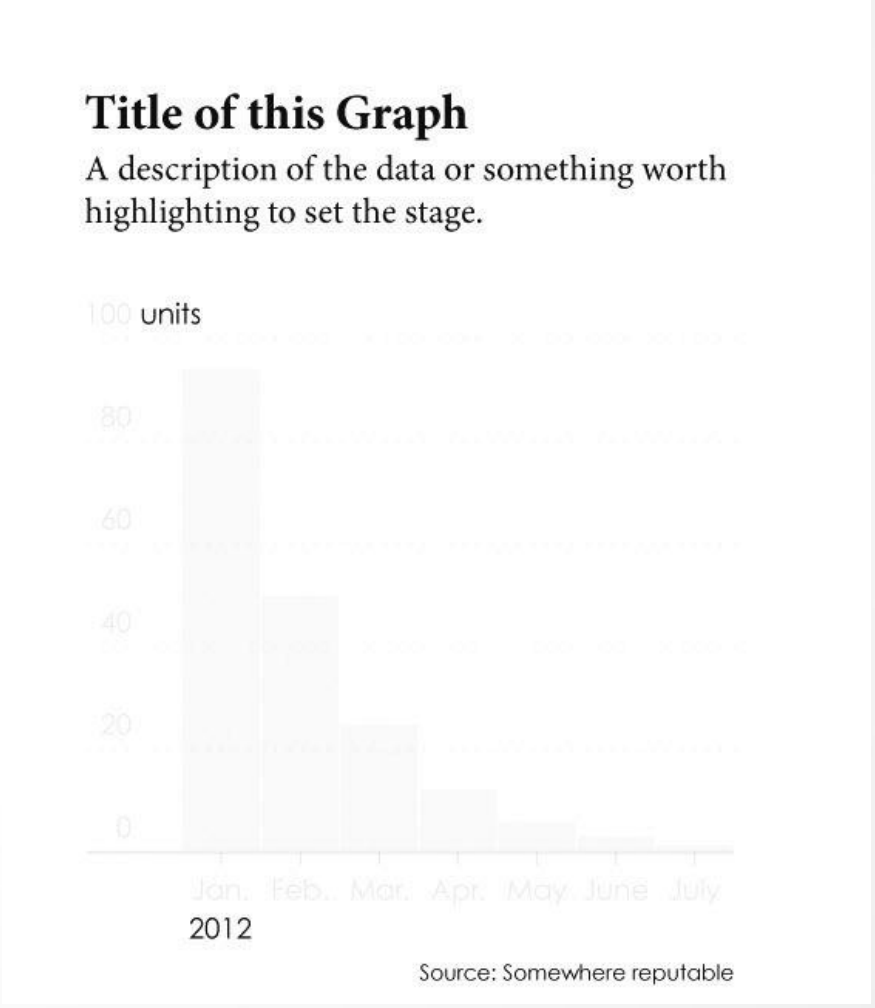
INTRO VISUALIZ. DE DATOS - ESTRUCTURA DE UN GRÁFICO ESTADÍSTICO

- **Sistema de coordenadas**: El sistema de coordenadas utilizado será uno u otro en función del gráfico a representar. No es lo mismo representar un gráfico de tarta, que un gráfico de dispersión, pej.



INTRO VISUALIZ. DE DATOS - ESTRUCTURA DE UN GRÁFICO ESTADÍSTICO

- **Contexto**: Si la audiencia no esta familiarizada con los datos, Para ambientes no familiarizados con los datos, el contexto



INTRO VISUALIZ. DE DATOS - PYLAB

- Python dispone de múltiples módulos para graficar y visualizar datos.
- Entre otros podemos encontrar:
  - matplotlib
  - pylab
  - Gleam (similar a shiny en R)
  - Bokeh (librería interactiva para visualizar datos en exploradores web)
  - ...

- Nos centraremos en el módulo **pylab**

INTRO VISUALIZ. DE DATOS - PYLAB

- Por defecto, pylab, viene instalado con Anaconda.
  En caso de no estar instalado, deberemos instalarlo siguiendo los pasos descritos en el punto 8 Trabajar con ficheros – Excel's.

- Importar el módulo.

```
import pylab as plt
```

- Acceso a los procedimientos de graficado y ploteo del módulo.

```
plt.<nombre_del_procedimiento>
```

- Dibujo básico (x e y tienen el mismo número de registros o son de la misma longitud).

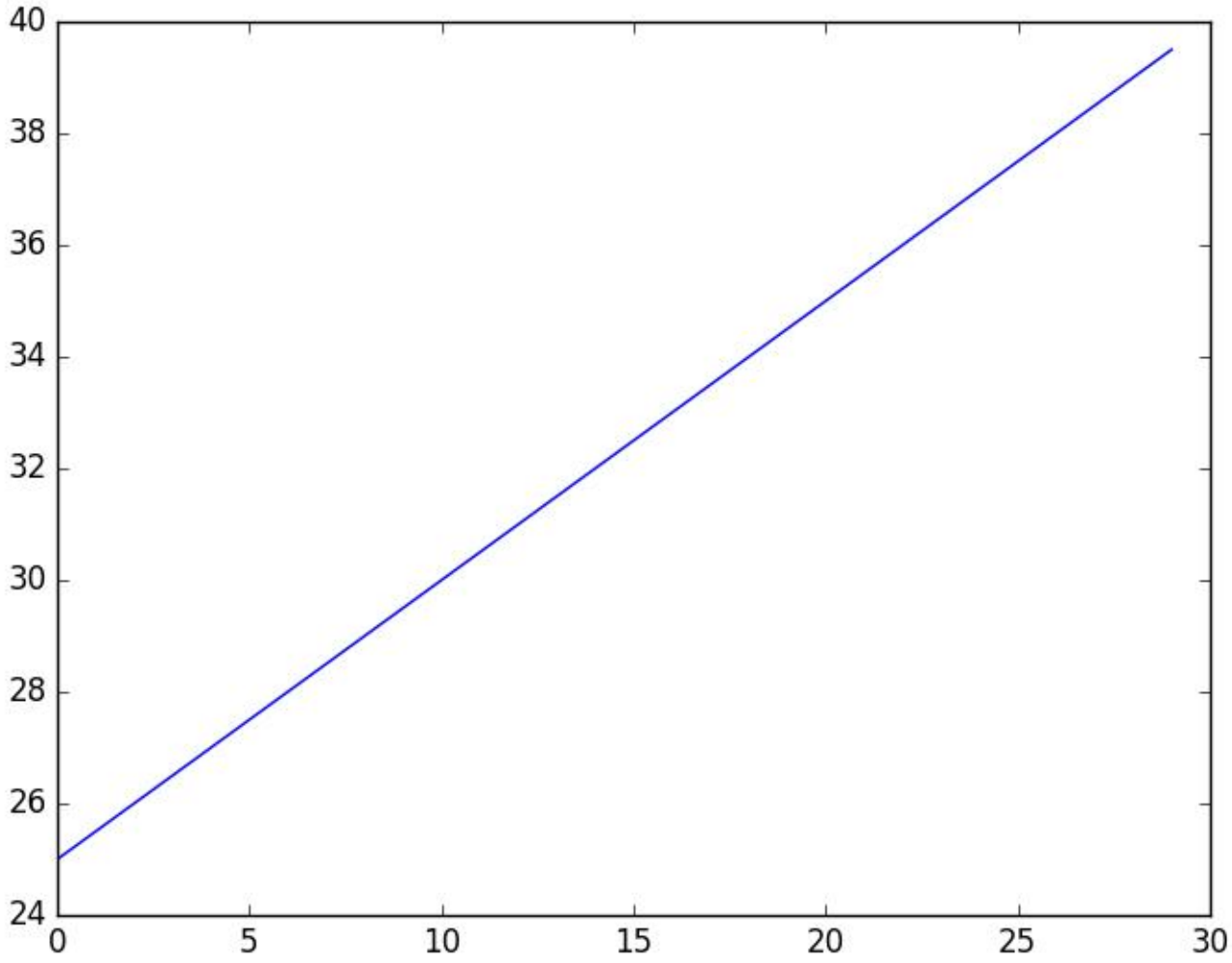
```
plt.plot(datos_eje_x, datos_eje_y)
```

INTRO VISUALIZ. DE DATOS - PYLAB

•Ejemplo:

```
import pylab as plt

x = []                      # creamos una lista vacía para
                            # los datos del eje x
ylineal = []                # creamos una lista vacía para
                            # los datos del eje y

for a in range (30):  # Rellenamos los datos de x e y
    x.append(a)
    ylineal.append(25 + 0.5*a)


plt.plot(x,ylineal)
```

INTRO VISUALIZ. DE DATOS - PYLAB



INTRO VISUALIZ. DE DATOS - PYLAB

•Superponer diferentes gráficos sobre el mismo gráfico. Por defecto, Pylab, pinta todos los gráficos en un mismo marco.

```
x = []
ylineal = []
ycuadrado = []
ycubo = []
yexponencial = []

for a in range (30):
    x.append(a)
    ylineal.append(25+0.5*a)
    ycuadrado.append(a**2)
    ycubo.append(a**3)
    yexponencial.append(1.5**a)

plt.plot(x,ylineal)
plt.plot(x,ycuadrado)
plt.plot(x,ycubo)
plt.plot(x,yexponencial)
```

INTRO VISUALIZ. DE DATOS - PYLAB

- Dibujar cada gráfico en su propio marco.

```
plt.figure(<argumento>)
```

```
plt.figure('Lineal')
plt.plot(x,ylineal)

plt.figure('Cuadrado')
plt.plot(x,ycuadrado)

plt.figure('Cubo')
plt.plot(x,ycubo)

plt.figure('Exponencial')
plt.plot(x,yexponencial)
```

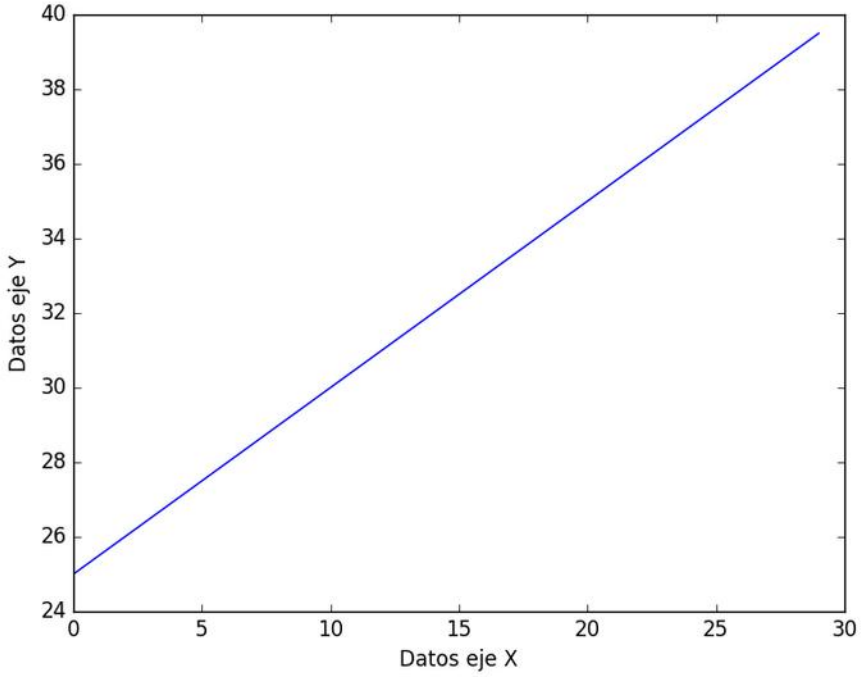
INTRO VISUALIZ. DE DATOS - PYLAB

- **Añadir nombres a los ejes.**

  Este comando, junto con todos los que añaden elementos a los gráficos, deben ejecutarse después de haber definido el marco donde se pintará el gráfico.

```
plt.xlabel(<nombre>)
plt.ylabel(<nombre>)
```

```
plt.xlabel('Datos eje X')
plt.ylabel('Datos eje Y')
```

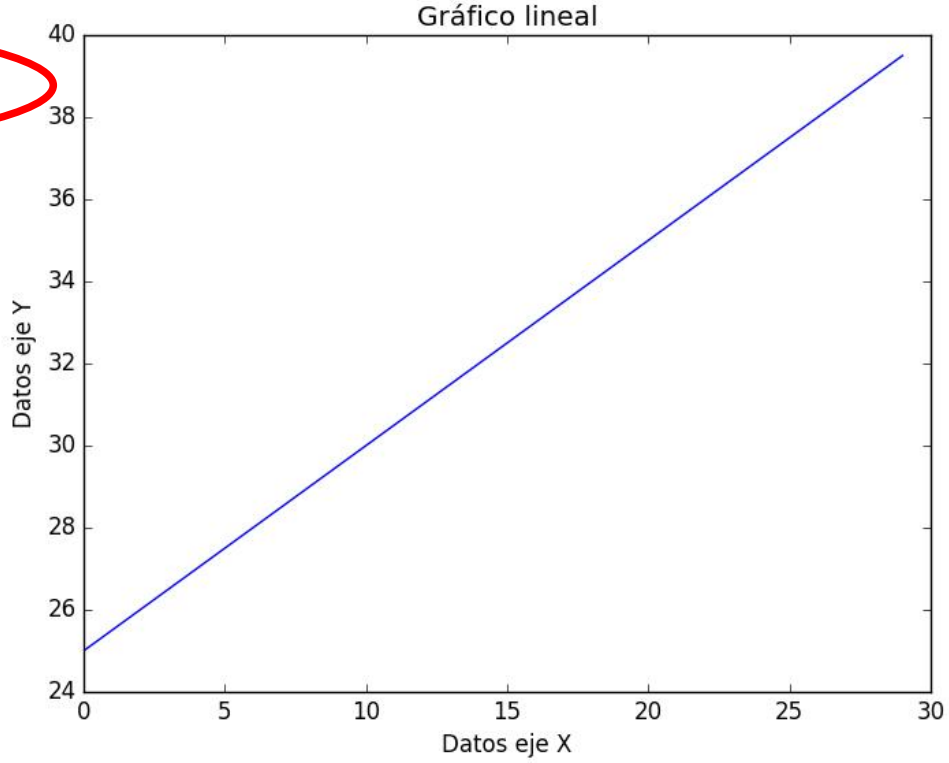


INTRO VISUALIZ. DE DATOS - PYLAB

- **Añadir título al gráfico.**

  Este comando debe ir después del comando que define el marco para el gráfico.

```
plt.figure('Lineal')

plt.title('Gráfico Lineal')
```



INTRO VISUALIZ. DE DATOS - PYLAB

- **Limpiar el marco de trabajo.**

  Este comando debe ir después del comando que define el marco para el gráfico y obliga a Python a redibujar el gráfico.

  Si hacemos diferentes pruebas de gráficos sobre un mismo marco, puede que los resultados no sean los esperados y conviene limpiar dicho marco antes de graficar.

```
plt.clf()
```

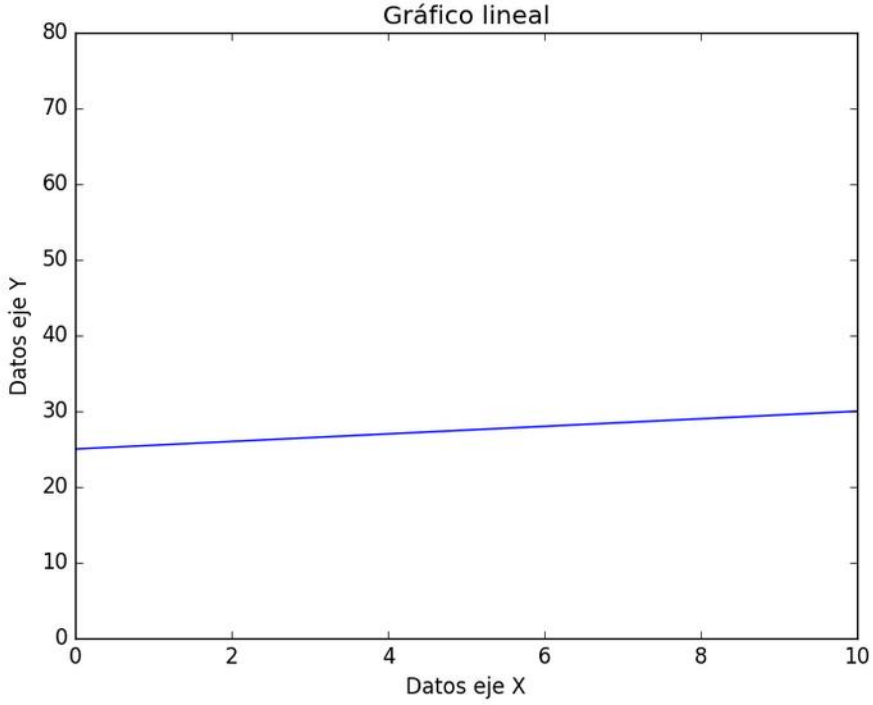
INTRO VISUALIZ. DE DATOS - PYLAB

- **Añadir escalas a los ejes.**

  Este comando debe ir después del comando que define el marco para el gráfico.

```
plt.xlim (lim_inferior, lim_superior)
plt.ylim (lim_inferior, lim_superior)
```

```
plt.xlim(0,10)
plt.ylim(0,80)
```

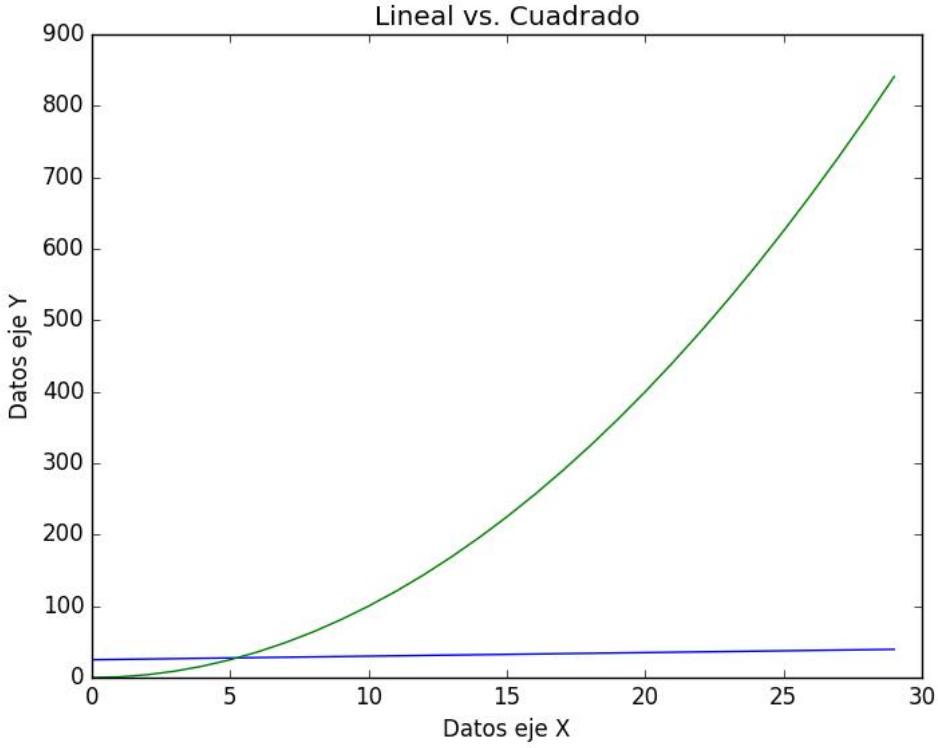


INTRO VISUALIZ. DE DATOS - PYLAB

- **Dibujar 2 gráficos en el mismo marco**

  Este comando debe ir después del comando que define el marco para el gráfico.

```
plt.figure('Lineal y Cuadrado')
plt.title("Lineal vs. Cuadrado")
plt.xlabel("Datos eje X")
plt.ylabel("Datos eje Y")
plt.plot(x,ylineal)
plt.plot(x,ycuadrado)
```



INTRO VISUALIZ. DE DATOS - PYLAB

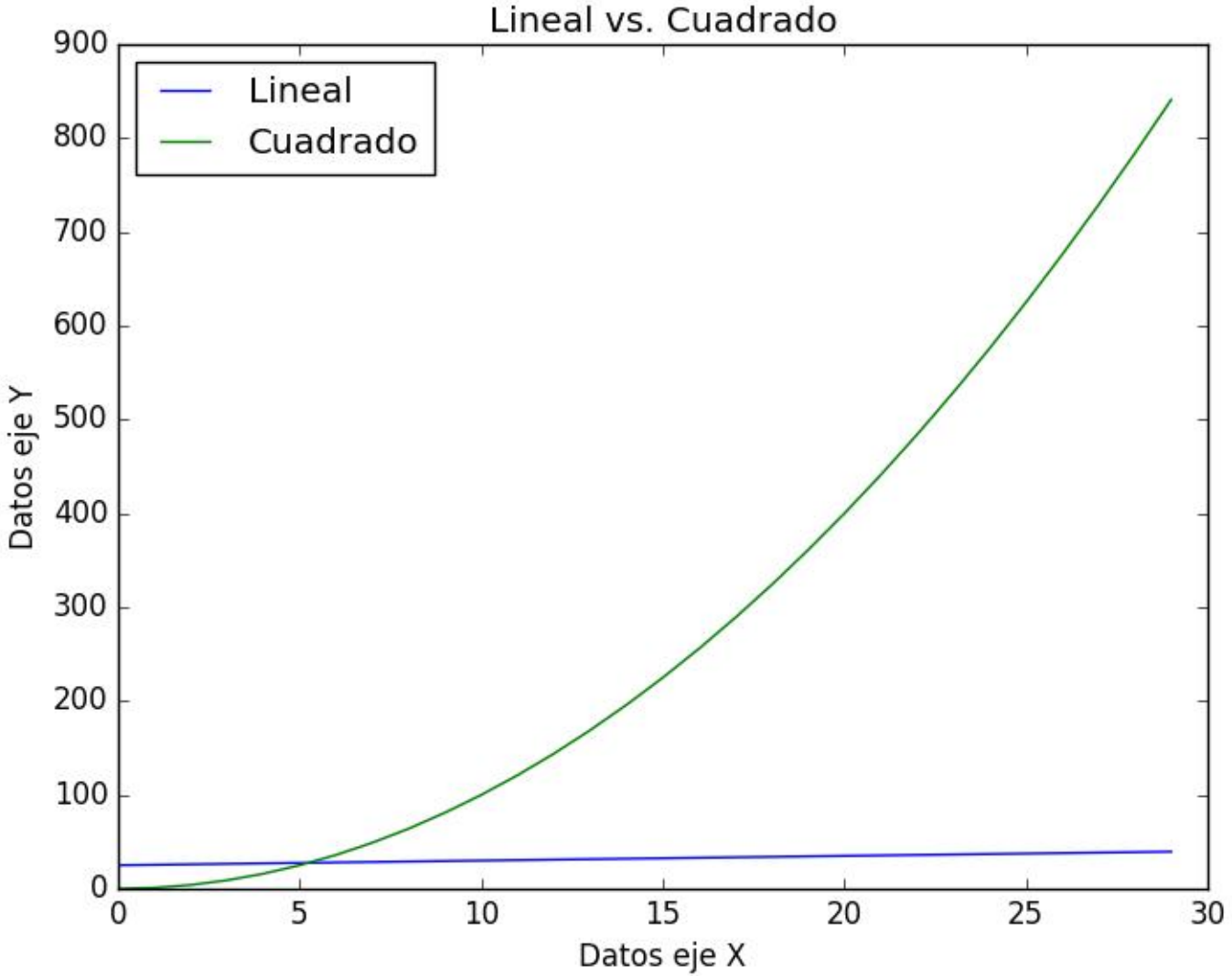
- **Añadir leyenda al gráfico**

  Este comando debe ir después del comando que define el marco para el gráfico.

```
plt.figure('Lineal y Cuadrado')
plt.clf()
plt.title("Lineal vs. Cuadrado")
plt.xlabel("Datos eje X")
plt.ylabel("Datos eje Y")
plt.plot(x,ylineal, label = "Lineal")
plt.plot(x,ycuadrado, label = "Cuadrado")
plt.legend(loc="upper left")
```

INTRO VISUALIZ. DE DATOS - PYLAB

- Añadir leyenda al gráfico



INTRO VISUALIZ. DE DATOS - PYLAB

- **Definir los parámetros de visualización.**
  - Modificar el color y tipo de datos del dataset
  - Modificar el ancho de lo dibujado
  - Visualizar varios plots
  - ...

```
plt.figure('Lineal y Cuadrado')
plt.clf()
plt.title("Lineal vs. Cuadrado")
plt.xlabel("Datos eje X")
plt.ylabel("Datos eje Y")
plt.plot(x,ylineal, "b-", label = "Lineal")
plt.plot(x,ycuadrado, "ro", label = "Cuadrado")
plt.legend(loc="upper left")
```

INTRO VISUALIZ. DE DATOS - PYLAB

- Definir los parámetros de visualización.

  - El primer elemento, corresponde al color de la línea (**`b = blue, k = black`**).
  - El segundo elemento, define el estilo de la línea.

Ejemplos:

```
'b'    # Línea con marcadores por defecto azules.
'ro'   # Línea roja, con marcadores circulares.
'g-'   # Línea verde sólida.
'--'   #  Línea discontinua, con color por defecto
'k^:'  # Línea negra, con triángulos como marcadores, conectados por
       una línea de puntos discontinua.
```

INTRO VISUALIZ. DE DATOS - PYLAB

- Definir los parámetros de visualización.

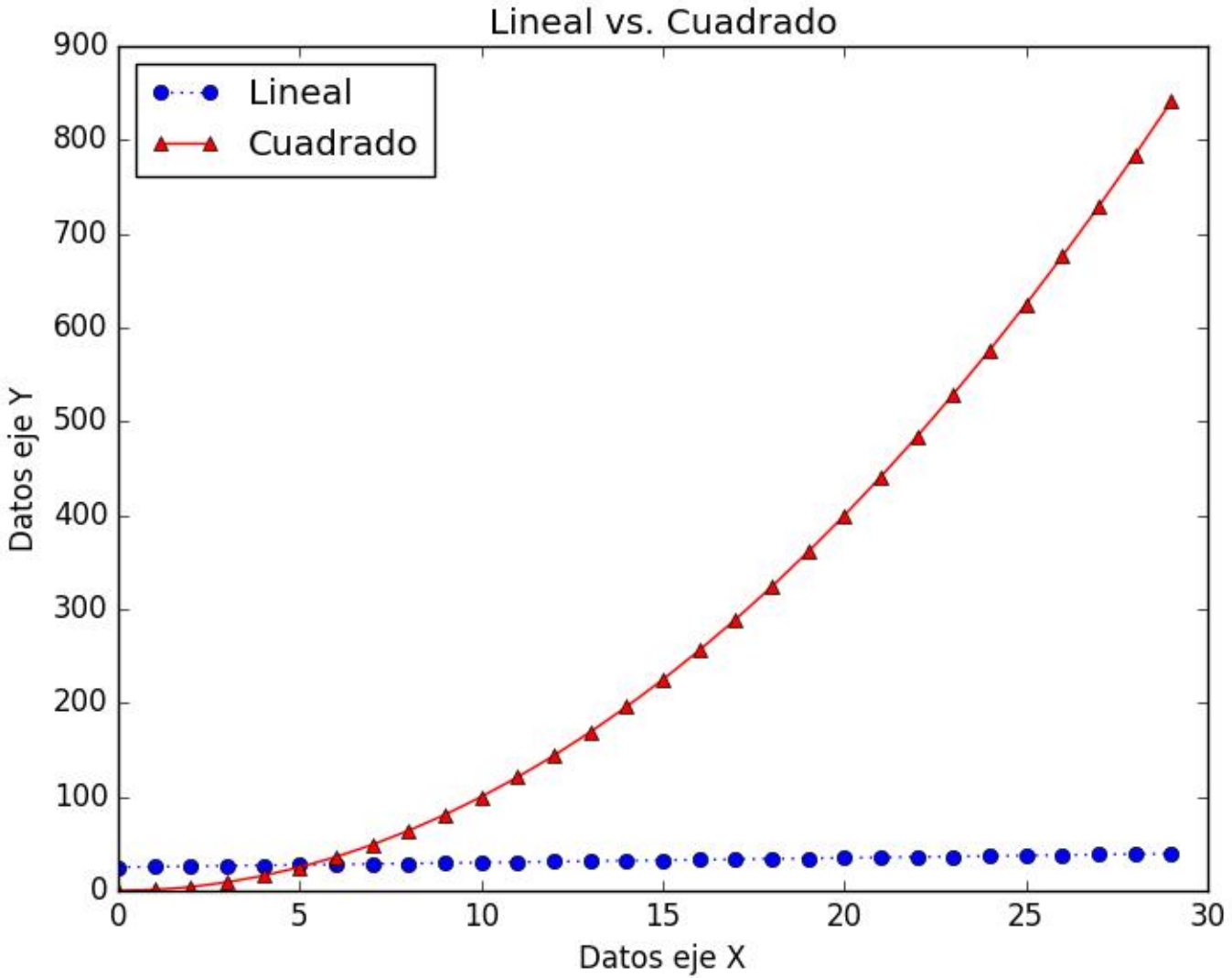
Parámetros color	
Caracter	Color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Parámetros tipo de línea	
Carácter	Descripción
-	Línea sólida
--	Línea discontinua
-.	Línea discontinua por puntos y rayas.
:	Línea compuesta de puntos

Algunos parámetros tipo de marcador	
Carácter	Descripción
.	Punto como marcador
o	Círculo
v	Triángulo invertido
^	Triángulo

INTRO VISUALIZ. DE DATOS - PYLAB

- Definir los parámetros de visualización.



INTRO VISUALIZ. DE DATOS - PYLAB

- Definir los parámetros de visualización.
  - Cambio del ancho de la línea:

```
plt.plot(x,ylineal, 'b:o', label = "Lineal",
linewidth = 2.0)
```

INTRO VISUALIZ. DE DATOS - PYLAB

- Visualizar varios plots de manera independiente en un mismo marco.
  - En este caso, debemos asegurarnos que los plots tienen la misma escala (al menos en uno de sus ejes) para poder compararlos.

```
plt.subplot (XYZ)
```

  - **X = Número de filas**
  - **Y = Número de columnas**
  - **Z = Ubicación del gráfico**

INTRO VISUALIZ. DE DATOS - PYLAB

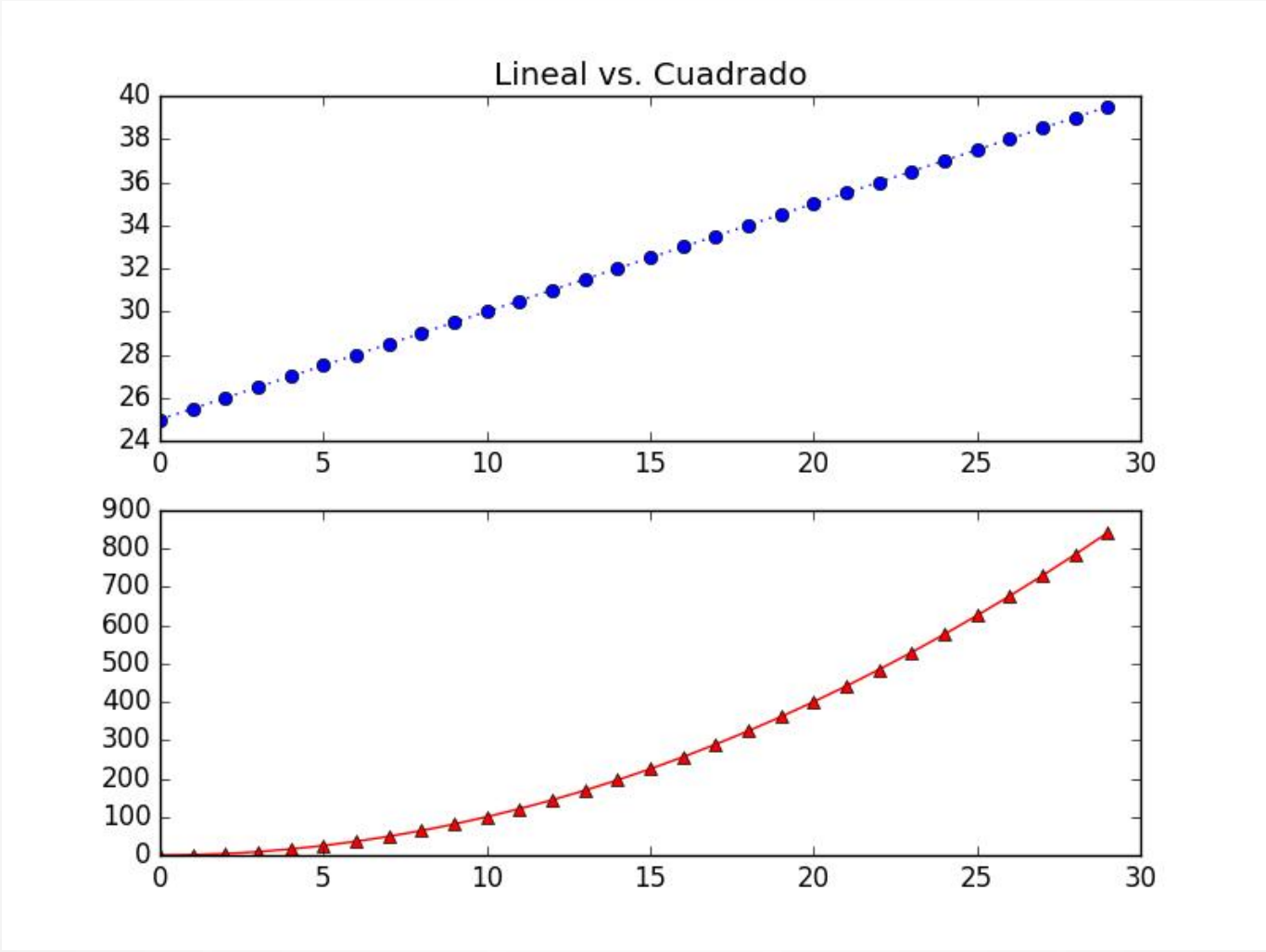
- Visualizar varios plots de manera independiente en un mismo marco.

```
plt.figure('Lineal y Cuadrado')
plt.clf()

plt.subplot(211)
plt.title("Lineal vs. Cuadrado")
plt.plot(x,ylineal, 'b:o', label = "Lineal", )

plt.subplot(212)
plt.plot(x,ycuadrado, 'r-^', label = "Cuadrado")
```

- Subplot (211) → coloca el plot en un marco de 2 filas x 1 columna, en la primera posición.
- Subplot (212) → coloca el plot en un marco de 2 filas x 1 columna, en la segunda posición.

INTRO VISUALIZ. DE DATOS - PYLAB

- Visualizar varios plots de manera independiente en un mismo marco.



INTRO VISUALIZ. DE DATOS - PYLAB

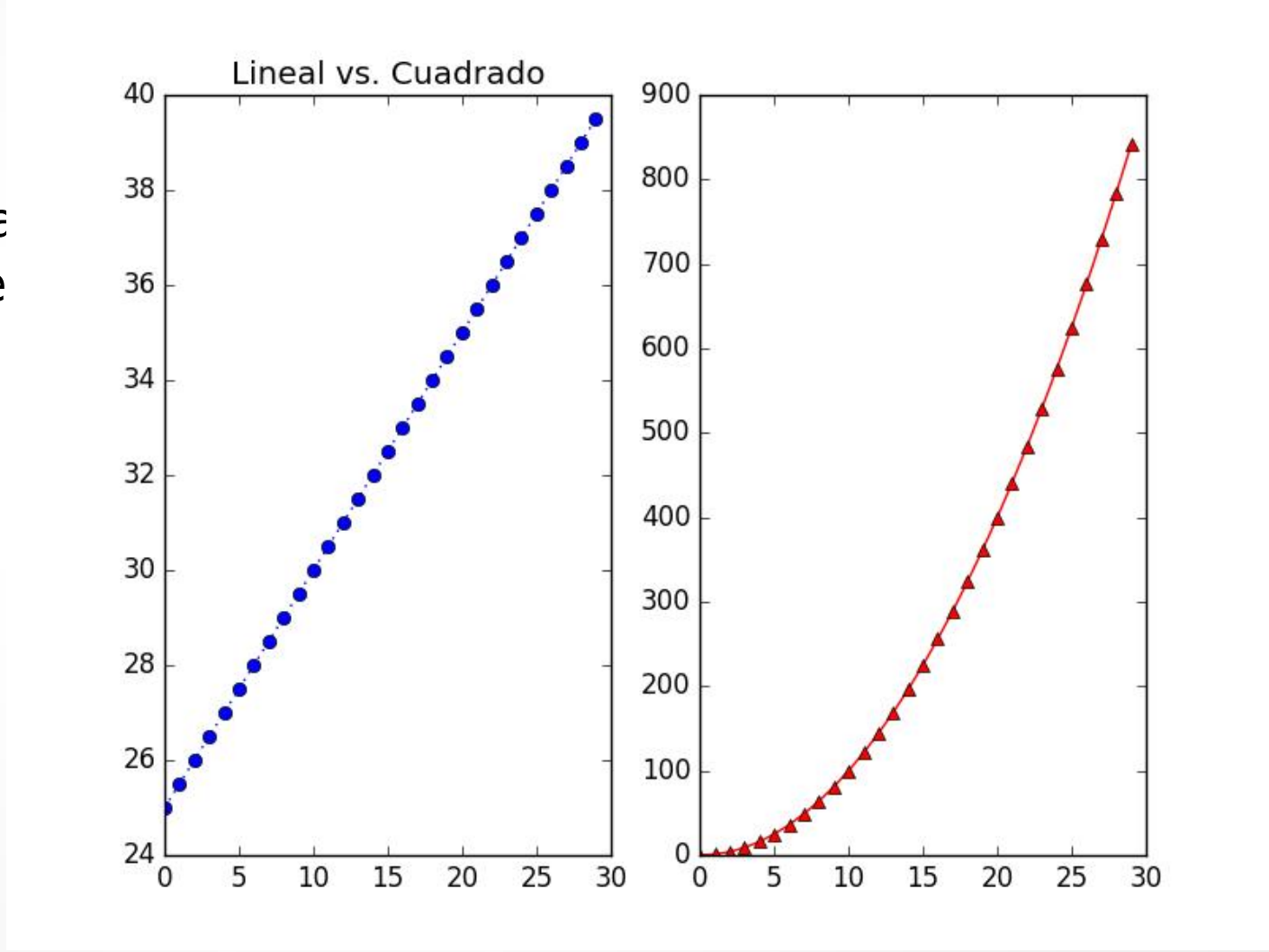
- Visualizar varios plots de manera independiente en un mismo marco.

```
plt.figure('Lineal y Cuadrado')
plt.clf()

plt.subplot(121)
plt.title("Lineal vs. Cuadrado")
plt.plot(x,ylineal, 'b:o', label = "Lineal", )

plt.subplot(122)
plt.plot(x,ycuadrado, 'r-^', label = "Cuadrado")
```

INTRO VISUALIZ. DE DATOS - PYLAB

- Visualizar varios plots de manera independiente en un mismo marco.

- En este caso, no podemos compara los gráficos, ya que e eje Y no tiene la misma escala para cada gráfico.



INTRO VISUALIZ. DE DATOS - PYLAB

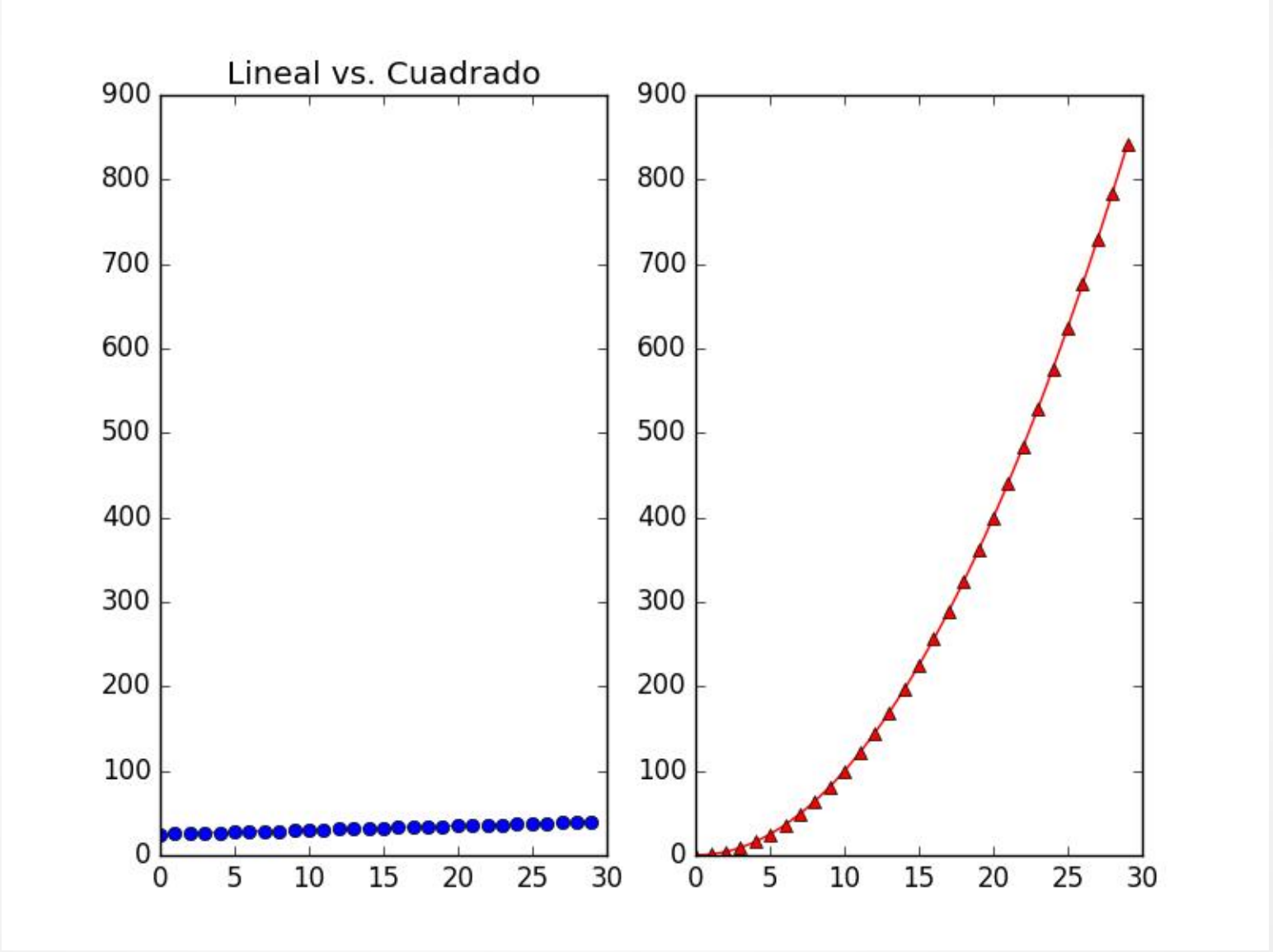
- Visualizar varios plots de manera independiente en un mismo marco.

→ Podemos “arreglar” el ejemplo anterior, definiendo la escala del eje Y para los plots.

```
plt.figure('Lineal y Cuadrado')
plt.clf()

plt.subplot(121)
plt.title("Lineal vs. Cuadrado")
plt.ylim(0,900)
plt.plot(x,ylineal, 'b:o', label = "Lineal", )

plt.subplot(122)
plt.plot(x,ycuadrado, 'r-^', label = "Cuadrado")
plt.ylim(0,900)
```

INTRO VISUALIZ. DE DATOS - PYLAB

- Visualizar varios plots de manera independiente en un mismo marco.



AYUDA ONLINE

- **Algunas webs con contenido Python:**
  - https://docs.python.org/3/
  - http://librosweb.es/libro/python/
  - https://www.pythoncheatsheet.org/
- **Ayuda sobre métodos principales extendida:**
  - Cadenas → http://librosweb.es/libro/python/capitulo_6.html
  - Listas → http://librosweb.es/libro/python/capitulo_7.html
  - Diccionarios → http://librosweb.es/libro/python/capitulo_8.html
  - Y muchas más...

Germán Alonso Lascurain
germanalonso@opendeusto.es