

# Manual de Python

Alfredo Sánchez Alberca  
([asalber@ceu.es](mailto:asalber@ceu.es))

Febrero 2020



---

## Términos de la licencia

Esta obra está bajo una licencia Atribución–No comercial–Compartir igual 4.0 Internacional de Creative Commons. Para ver una copia de esta licencia, visite <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>.

Con esta licencia eres libre de:

- **Compatir:** Copiar y redistribuir el material en cualquier medio o formato
- **Adaptar:** Remezclar, transformar y construir a partir del material

Bajo las siguientes términos:



**Atribución.** Usted debe dar crédito de manera adecuada, brindar un enlace a la licencia, e indicar si se han realizado cambios. Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo de la licenciante.



**No comercial.** Usted no puede hacer uso del material con propósitos comerciales.



**Compartir igual.** Si remezcla, transforma o crea a partir del material, debe distribuir su contribución bajo la misma licencia del original.

No hay restricciones adicionales — No puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otras a hacer cualquier uso permitido por la licencia.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
  - alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor
  - Nada en esta licencia menoscaba o restringe los derechos morales del autor.
-

## Índice general

<b>1</b>	<b>Introducción a Python</b>	<b>7</b>
1.1	¿Qué es Python?	7
1.2	Principales ventajas de Python	7
1.3	Tipos de ejecución	7
1.3.1	Interpretado en la consola de Python	7
1.3.2	Interpretado en fichero	7
1.3.3	Compilado a bytecode	8
1.3.4	Compilado a ejecutable del sistema	8
<b>2</b>	<b>Tipos de datos simples</b>	<b>8</b>
2.1	Tipos de datos primitivos simples	8
2.2	Tipos de datos primitivos compuestos (contenedores)	9
2.3	Clase de un dato ( <code>type()</code> )	9
2.4	Números (clases <code>int</code> y <code>float</code> )	9
2.4.1	Operadores aritméticos	10
2.4.2	Operadores lógicos con números	10
2.5	Cadenas (clase <code>str</code> )	11
2.5.1	Acceso a los elementos de una cadena	11
2.5.2	Subcadenas	12
2.5.3	Operaciones con cadenas	12
2.5.4	Operaciones de comparación de cadenas	12
2.5.5	Funciones de cadenas	13
2.5.6	Cadenas formateadas ( <code>format()</code> )	13
2.6	Datos lógicos o booleanos (clase <code>bool</code> )	14
2.6.1	Operaciones con valores lógicos	14
2.6.2	Tabla de verdad	15
2.7	Conversión de datos primitivos simples	15
2.8	Variables	16
2.9	Entrada por terminal ( <code>input()</code> )	16
2.9.1	Salida por terminal ( <code>print()</code> )	17
<b>3</b>	<b>Estructuras de control</b>	<b>17</b>
3.1	Condicionales ( <code>if</code> )	17
3.2	Bucles condicionales ( <code>while</code> )	18
3.3	Bucles iterativos ( <code>for</code> )	19
<b>4</b>	<b>Tipos de datos estructurados</b>	<b>19</b>
4.1	Listas	19
4.1.1	Creación de listas mediante la función <code>list()</code>	20
4.1.2	Acceso a los elementos de una lista	20
4.1.3	Sublistas	21

4.1.4	Operaciones que no modifican una lista . . . . .	21
4.1.5	Operaciones que modifican una lista . . . . .	22
4.1.6	Copia de listas . . . . .	23
4.2	Tuplas . . . . .	23
4.2.1	Creación de tuplas mediante la función <code>tuple()</code> . . . . .	24
4.2.2	Operaciones con tuplas . . . . .	24
4.3	Diccionarios . . . . .	25
4.3.1	Acceso a los elementos de un diccionario . . . . .	25
4.3.2	Operaciones que no modifican un diccionario . . . . .	26
4.3.3	Operaciones que modifican un diccionario . . . . .	26
4.3.4	Copia de diccionarios . . . . .	27
<b>5</b>	<b>Funciones</b>	<b>28</b>
5.1	Funciones ( <code>def</code> ) . . . . .	28
5.1.1	Parámetros de una función . . . . .	28
5.1.2	Argumentos de la llamada a una función . . . . .	28
5.1.3	Retorno de una función . . . . .	29
5.2	Argumentos por defecto . . . . .	29
5.3	Pasar un número indeterminado de argumentos . . . . .	29
5.4	Ámbito de los parámetros y variables de una función . . . . .	30
5.5	Ámbito de los parámetros y variables de una función . . . . .	30
5.6	Paso de argumentos por referencia . . . . .	31
5.7	Documentación de funciones . . . . .	31
5.8	Funciones recursivas . . . . .	32
5.8.1	Funciones recursivas múltiples . . . . .	32
5.8.2	Los riesgos de la recursión . . . . .	32
5.9	Importación de funciones ( <code>import</code> ) . . . . .	33
5.9.1	Módulos de la librería estándar más importantes . . . . .	33
5.9.2	Otros módulos imprescindibles . . . . .	34
5.10	Programación funcional . . . . .	34
5.10.1	Funciones anónimas ( <code>lambda</code> ) . . . . .	34
5.10.2	Aplicar una función a todos los elementos de una colección iterable ( <code>map</code> ) . . . . .	35
5.10.3	Filtrar los elementos de una colección iterable ( <code>filter</code> ) . . . . .	35
5.10.4	Combinar los elementos de varias colecciones iterables ( <code>zip</code> ) . . . . .	35
5.10.5	Operar todos los elementos de una colección iterable ( <code>reduce</code> ) . . . . .	36
5.11	Comprensión de colecciones . . . . .	36
5.11.1	Comprensión de listas . . . . .	36
5.11.2	Comprensión de diccionarios . . . . .	37
<b>6</b>	<b>Ficheros y excepciones</b>	<b>37</b>
6.1	Ficheros . . . . .	37
6.1.1	Creación y escritura de ficheros . . . . .	37
6.1.2	Añadir datos a un fichero . . . . .	38

6.1.3	Leer datos de un fichero . . . . .	38
6.1.4	Leer datos de un fichero . . . . .	38
6.1.5	Cerrar un fichero . . . . .	39
6.1.6	Renombrado y borrado de un fichero . . . . .	39
6.1.7	Renombrado y borrado de un fichero o directorio . . . . .	39
6.1.8	Creación, cambio y eliminación de directorios . . . . .	40
6.1.9	Leer un fichero de internet . . . . .	40
6.2	Control de errores mediante excepciones . . . . .	40
6.2.1	Tipos de excepciones . . . . .	41
6.2.2	Control de excepciones . . . . .	41
6.2.3	Control de excepciones . . . . .	41
<b>7</b>	<b>La librería Numpy</b>	<b>42</b>
7.1	La librería NumPy . . . . .	42
7.2	La clase de objetos array . . . . .	42
7.3	Creación de arrays . . . . .	43
7.3.1	Atributos de un array . . . . .	44
7.3.2	Acceso a los elementos de un array . . . . .	44
7.3.3	Filtrado de elementos de un array . . . . .	45
7.3.4	Operaciones matemáticas con arrays . . . . .	45
7.3.5	Operaciones matemáticas a nivel de array . . . . .	45
<b>8</b>	<b>La librería Pandas</b>	<b>46</b>
8.1	La librería Pandas . . . . .	46
8.1.1	La clase de objetos DataFrame . . . . .	46
8.1.2	Creación de un DataFrame a partir de un diccionario . . . . .	47
8.1.3	Importación de ficheros . . . . .	47
8.1.4	Exportación de ficheros . . . . .	48
8.1.5	Atributos de un DataFrame . . . . .	48
8.1.6	Acceso a los elementos de un DataFrame mediante índices . . . . .	48
8.1.7	Acceso a los elementos de un DataFrame mediante nombres . . . . .	49
8.1.8	Operaciones sobre columnas . . . . .	49
8.1.9	Aplicar funciones a columnas . . . . .	50
8.1.10	Filtrado de un DataFrame . . . . .	50
8.2	Eliminar filas y columnas de un DataFrame . . . . .	50
8.2.1	Ordenar un DataFrame . . . . .	51
8.2.2	Reestructurar un DataFrame: Convertir columnas en filas . . . . .	52
8.2.3	Reestructurar un DataFrame: Convertir filas en columnas . . . . .	52
<b>9</b>	<b>La librería Matplotlib</b>	<b>53</b>
9.1	La librería Matplotlib . . . . .	53
9.1.1	Tipos de gráficos . . . . .	53
9.1.2	Creación de gráficos con matplotlib . . . . .	53
9.1.3	Diagramas de dispersión o puntos . . . . .	55

9.1.4	Diagramas de líneas . . . . .	56
9.1.5	Diagramas de areas . . . . .	57
9.1.6	Diagramas de barras verticales . . . . .	58
9.1.7	Diagramas de barras horizontales . . . . .	59
9.1.8	Histogramas . . . . .	60
9.1.9	Diagramas de sectores . . . . .	61
9.1.10	Diagramas de caja y bigotes . . . . .	62
9.1.11	Diagramas de violín . . . . .	63
9.1.12	Diagramas de contorno . . . . .	64
9.1.13	Mapas de color . . . . .	65
<b>10</b>	<b>Apéndice: Depuración de código</b>	<b>67</b>
10.1	Depuración de programas . . . . .	67
10.1.1	Comandos de depuración . . . . .	67
10.1.2	Depuración en Visual Studio Code . . . . .	68
10.2	Referencias . . . . .	70

## 1 Introducción a Python

### 1.1 ¿Qué es Python?

Python es un lenguaje de programación de alto nivel multiparadigma que permite:

- Programación imperativa
- Programación funcional
- Programación orientada a objetos

Fue creado por Guido van Rossum en 1990 aunque actualmente es desarrollado y mantenido por la [Python Software Foundation](#)

### 1.2 Principales ventajas de Python

- Es de código abierto (certificado por la OSI).
- Es interpretable y compilable.
- Es fácil de aprender gracias a que su sintaxis es bastante legible para los humanos.
- Es un lenguaje maduro (29 años).
- Es fácilmente extensible e integrable en otros lenguajes (C, java).
- Esta mantenido por una gran comunidad de desarrolladores y hay multitud de recursos para su aprendizaje.

### 1.3 Tipos de ejecución

#### 1.3.1 Interpretado en la consola de Python

Se ejecuta cada instrucción que introduce el usuario de manera interactiva.

```
1 > python
2 >>> name = "Alf"
3 >>> print("Hola ", name)
4 Hola Alf
```

#### 1.3.2 Interpretado en fichero

Se leen y se ejecutan una a una todas las instrucciones del fichero.

```
1 # Fichero hola.py
2 name = "Alf"
3 print("Hola ", name)
```

```
1 > python hola.py
2 Hola Alf
```

También se puede hacer el fichero ejecutable indicando en la primera línea la ruta hasta el intérprete de Python.

```
1 #!/usr/bin/python3
2 name = "Alf"
3 print("Hola", name)
```

```
1 > chmod +x hola.py
2 > ./hola.py
3 Hola Alf
```

### 1.3.3 Compilado a bytecode

```
1 # Fichero hola.py
2 name = "Alf"
3 print("Hola " + name)
```

```
1 > python -O -m py_compile hola.py
2 > python __pycache__/hola.cpython-37.pyc
3 Hola Alf
```

### 1.3.4 Compilado a ejecutable del sistema

Hay distintos paquetes que permiten compilar a un ejecutable del sistema operativo usado, por ejemplo `pyinstaller`.

```
1 > conda install pyinstaller
2 > pyinstaller hola.py
3 > ./dist/hola/hola
4 Hola Alf
```

## 2 Tipos de datos simples

### 2.1 Tipos de datos primitivos simples

- **Números** (numbers): Secuencia de dígitos (pueden incluir el - para negativos y el . para decimales) que representan números.

**Ejemplo.** 0, -1, 3.1415.



- **Cadenas** (strings): Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas simples o dobles.

**Ejemplo.** 'Hola', "Adiós".

- **Booleanos** (boolean): Contiene únicamente dos elementos `True` y `False` que representan los valores lógicos verdadero y falso respectivamente.

Estos datos son inmutables, es decir, su valor es constante y no puede cambiar.

## 2.2 Tipos de datos primitivos compuestos (contenedores)

- **Listas** (lists): Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. Se representan con corchetes y los elementos se separan por comas.

**Ejemplo.** [1, "dos", [3, 4], True].

- **Tuplas** (tuples). Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. A diferencia de las listas son inmutables, es decir, que no cambian durante la ejecución. Se representan mediante paréntesis y los elementos se separan por comas.

**Ejemplo.** (1, 'dos', 3)

- **Diccionarios** (dictionaries): Colecciones de objetos con una clave asociada. Se representan con llaves, los pares separados por comas y cada par contiene una clave y un objeto asociado separados por dos puntos.

**Ejemplo.** {'pi':3.1416, 'e':2.718}.

## 2.3 Clase de un dato (type())

La clase a la que pertenece un dato se obtiene con el comando `type()`

```
1 >>> type(1)
2 <class 'int'>
3 >>> type("Hola")
4 <class 'str'>
5 >>> type([1, "dos", [3, 4], True])
6 <class 'list'>
7 >>> type({'pi':3.1416, 'e':2.718})
8 <class 'dict'>
9 >>> type((1, 'dos', 3))
10 <class 'tuple'>
```

## 2.4 Números (clases int y float)

Secuencia de dígitos (pueden incluir el - para negativos y el . para decimales) que representan números. Pueden ser enteros (`int`) o reales (`float`).

```
1 >>> type(1)
2 <class 'int'>
```

```
3 >>> type(-2)
4 <class 'int'>
5 >>> type(2.3)
6 <class 'float'>
```

### 2.4.1 Operadores aritméticos

- Operadores aritméticos: + (suma), - (resta), \* (producto), / (cociente), // (cociente división entera), % (resto división entera), \*\* (potencia).

Orden de prioridad de evaluación:

- 
- 1 Funciones predefinidas
  - 2 Potencias
  - 3 Productos y cocientes
  - 4 Sumas y restas
- 

Se puede saltar el orden de evaluación utilizando paréntesis ( ).

```
1 >>> 2+3
2 5
3 >>> 5*-2
4 -10
5 >>> 5/2
6 2.5
7 >>> 5//2
8 2
9 >>> (2+3)**2
10 25
```

### 2.4.2 Operadores lógicos con números

Devuelven un valor lógico o booleano.

- Operadores lógicos: == (igual que), > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que), != (distinto de).

```
1 >>> 3==3
2 True
3 >>> 3.1<=3
4 False
5 >>> -1!=1
```

```
6 True
```

## 2.5 Cadenas (clase str)

Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas sencillas ' o dobles ".

```
1 'Python'
2 "123"
3 'True'
4 # Cadena vacía
5 ''
6 # Cadena con un espacio en blanco
7 ' '
8 # Cambio de línea
9 '\n'
10 # Tabulador
11 '\t'
```

### 2.5.1 Acceso a los elementos de una cadena

Cada carácter tiene asociado un índice que permite acceder a él.

Cadena	P	y	t	h	o	n
Índice positivo	0	1	2	3	4	5
Índice negativo	-6	-5	-4	-3	-2	-1

- `c[i]` devuelve el carácter de la cadena `c` con el índice `i`.

*El índice del primer carácter de la cadena es 0.*

También se pueden utilizar índices negativos para recorrer la cadena del final al principio.

*El índice del último carácter de la cadena es -1.*

```
1 >>> 'Python'[0]
2 'P'
3 >>> 'Python'[1]
4 'y'
5 >>> 'Python'[-1]
6 'n'
7 >>> 'Python'[6]
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 IndexError: string index out of range
```

### 2.5.2 Subcadenas

- `c[i:j:k]` : Devuelve la subcadena de `c` desde el carácter con el índice `i` hasta el carácter anterior al índice `j`, tomando caracteres cada `k`.

```
1 >>> 'Python'[1:4]
2 'yth'
3 >>> 'Python'[1:1]
4 ''
5 >>> 'Python'[2:]
6 'thon'
7 >>> 'Python'[:-2]
8 'ytho'
9 >>> 'Python'[: ]
10 'Python'
11 >>> 'Python'[0:6:2]
12 'Pto'
```

### 2.5.3 Operaciones con cadenas

- `c1 + c2` : Devuelve la cadena resultado de concatenar las cadenas `c1` y `c2`.
- `c * n` : Devuelve la cadena resultado de concatenar `n` copias de la cadena `c`.
- `c1 in c2` : Devuelve `True` si `c1` es una cadena concenida en `c2` y `False` en caso contrario.
- `c1 not in c2` : Devuelve `True` si `c1` es una cadena no concenida en `c2` y `False` en caso contrario.

```
1 >>> 'Me gusta ' + 'Python'
2 'Me gusta Python'
3 >>> 'Python' * 3
4 'PythonPythonPython'
5 >>> 'y' in 'Python'
6 True
7 >>> 'tho' in 'Python'
8 True
9 >>> 'to' not in 'Python'
10 True
```

### 2.5.4 Operaciones de comparación de cadenas

- `c1 == c2` : Devuelve `True` si la cadena `c1` es igual que la cadena `c2` y `False` en caso contrario.
- `c1 > c2` : Devuelve `True` si la cadena `c1` sucede a la cadena `c2` y `False` en caso contrario.
- `c1 < c2` : Devuelve `True` si la cadena `c1` antecede a la cadena `c2` y `False` en caso contrario.
- `c1 >= c2` : Devuelve `True` si la cadena `c1` sucede o es igual a la cadena `c2` y `False` en caso contrario.
- `c1 <= c2` : Devuelve `True` si la cadena `c1` antecede o es igual a la cadena `c2` y `False` en caso contrario.
- `c1 != c2` : Devuelve `True` si la cadena `c1` es distinta de la cadena `c2` y `False` en caso contrario.

Utilizan el orden establecido en el *código ASCII*.

```
1 >>> 'Python' == 'python'
2 False
3 >>> 'Python' < 'python'
4 True
5 >>> 'a' > 'Z'
6 True
7 >>> 'A' >= 'Z'
8 False
9 >>> '' < 'Python'
10 True
```

### 2.5.5 Funciones de cadenas

- `len(c)` : Devuelve el número de caracteres de la cadena `c`.
- `min(c)` : Devuelve el carácter menor de la cadena `c`.
- `max(c)` : Devuelve el carácter mayor de la cadena `c`.
- `c.upper()` : Devuelve la cadena con los mismos caracteres que la cadena `c` pero en mayúsculas.
- `c.lower()` : Devuelve la cadena con los mismos caracteres que la cadena `c` pero en minúsculas.
- `c.title()` : Devuelve la cadena con los mismos caracteres que la cadena `c` con el primer carácter en mayúsculas y el resto en minúsculas.
- `c.split(delimitador)` : Devuelve la lista formada por las subcadenas que resultan de partir la cadena `c` usando como delimitador la cadena `delimitador`. Si no se especifica el delimitador utiliza por defecto el espacio en blanco.

```
1 >>> len('Python')
2 6
3 >>> min('Python')
4 'P'
5 >>> max('Python')
6 'y'
7 >>> 'Python'.upper()
8 'PYTHON'
9 >>> 'A,B,C'.split(',')
10 ['A', 'B', 'C']
11 >>> 'I love Python'.split()
12 ['I', 'love', 'Python']
```

### 2.5.6 Cadenas formateadas (`format()`)

- `c.format(valores)` : Devuelve la cadena `c` tras sustituir los valores de la secuencia `valores` en los marcadores de posición de `c`. Los marcadores de posición se indican mediante llaves `{}` en la cadena `c`, y el reemplazo de los valores se puede realizar por posición, indicando en número de orden del valor

dentro de las llaves, o por nombre, indicando el nombre del valor, siempre y cuando los valores se pasen con el formato `nombre = valor`.

```
1 >>> 'Un {} vale {} {}'.format('€', 1.12, '$')
2 'Un € vale 1.12 $'
3 >>> 'Un {2} vale {1} {0}'.format('€', 1.12, '$')
4 'Un $ vale 1.12 €'
5 >>> 'Un {moneda1} vale {cambio} {moneda2}'.format(moneda1 = '€', cambio
6           = 1.12, moneda2 = '$')
7 'Un € vale 1.12 $'
```

Los marcadores de posición, a parte de indicar la posición de los valores de reemplazo, pueden indicar también el formato de estos. Para ello se utiliza la siguiente sintaxis:

- `{:n}`: Alinea el valor a la izquierda rellenando con espacios por la derecha hasta los `n` caracteres.
- `{:>n}`: Alinea el valor a la derecha rellenando con espacios por la izquierda hasta los `n` caracteres.
- `{:^n}`: Alinea el valor en el centro rellenando con espacios por la izquierda y por la derecha hasta los `n` caracteres.
- `{:nd}`: Formatea el valor como un número entero con `n` caracteres rellenando con espacios blancos por la izquierda.
- `{:n.mf}`: Formatea el valor como un número real con un tamaño de `n` caracteres (incluido el separador de decimales) y `m` cifras decimales, rellenando con espacios blancos por la izquierda.

```
1 >>> 'Hoy es {:^10}, mañana {:10} y pasado {:>10}'.format('lunes', '
2     martes', 'miércoles')
3 'Hoy es   lunes   , mañana martes       y pasado  miércoles'
4 >>> 'Cantidad {:5d}'.format(12)
5 'Cantidad    12'
6 >>> 'Pi vale {:8.4f}'.format(3.141592)
7 'Pi vale    3.1416'
```

## 2.6 Datos lógicos o booleanos (clase `bool`)

Contiene únicamente dos elementos `True` y `False` que representan los valores lógicos verdadero y falso respectivamente.

`False` tiene asociado el valor 0 y `True` tiene asociado el valor 1.

### 2.6.1 Operaciones con valores lógicos

- Operadores lógicos: `==` (igual que), `>` (mayor), `<` (menor), `>=` (mayor o igual que), `<=` (menor o igual que), `!=` (distinto de).
- `not b` (negación): Devuelve `True` si el dato booleano `b` es `False`, y `False` en caso contrario.
- `b1 and b2`: Devuelve `True` si los datos booleanos `b1` y `b2` son `True`, y `False` en caso contrario.
- `b1 or b2`: Devuelve `True` si alguno de los datos booleanos `b1` o `b2` son `True`, y `False` en caso contrario.

### 2.6.2 Tabla de verdad

x	y	not x	x and y	x or y
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

```
1 >>> not True
2 False
3 >>> False or True
4 True
5 >>> True and False
6 False
7 >>> True and True
8 True
```

## 2.7 Conversión de datos primitivos simples

Las siguientes funciones convierten un dato de un tipo en otro, siempre y cuando la conversión sea posible.

- `int()` convierte a entero.  
**Ejemplo.** `int('12')` 12  
`int(True)` 1  
`int('c')` Error
- `float()` convierte a real.  
**Ejemplo.** `float('3.14')` 3.14  
`float(True)` 1.0  
`float('III')` Error
- `str()` convierte a cadena.  
**Ejemplo.** `str(3.14)` '3.14'  
`str(True)` 'True'
- `bool()` convierte a lógico.  
**Ejemplo.** `bool('0')` False  
`bool('3.14')` True  
`bool('')` False  
`bool('Hola')` True

## 2.8 Variables

Una variable es un identificador ligado a algún valor.

Reglas para nombrarlas:

- Comienzan siempre por una letra, seguida de otras letras o números.
- No se pueden utilizarse palabras reservadas del lenguaje.

A diferencia de otros lenguajes no tienen asociado un tipo y no es necesario declararlas antes de usarlas (tipado dinámico).

Para asignar un valor a una variable se utiliza el operador `=` y para borrar una variable se utiliza la instrucción `del`.

```
1 lenguaje = 'Python'
2 x = 3.14
3 y = 3 + 2
4 # Asignación múltiple
5 a1, a2 = 1, 2
6 # Intercambio de valores
7 a, b = b, a
8 # Incremento (equivale a x = x + 2)
9 x += 2
10 # Decremento (equivale a x = x - 1)
11 x -= 1
12 # Valor no definido
13 x = None
14 del x
```

## 2.9 Entrada por terminal (`input()`)

Para asignar a una variable un valor introducido por el usuario en la consola se utiliza la instrucción

`input(mensaje)` : Muestra la cadena `mensaje` por la terminal y devuelve una cadena con la entrada del usuario.

*El valor devuelto siempre es una cadena, incluso si el usuario introduce un dato numérico.*

```
1 >>> language = input('?Cuál es tu lenguaje favorito? ')
2 ?Cuál es tu lenguaje favorito? Python
3 >>> language
4 'Python'
5 >>> age = input('?Cuál es tu edad? ')
6 ?Cuál es tu edad? 20
7 >>> age
8 '20'
```



### 2.9.1 Salida por terminal (`print()`)

Para mostrar un dato por la terminal se utiliza la instrucción

```
print(dato1, ..., sep=' ', end='\n', file=sys.stdout)
```

donde

- `dato1, ...` son los datos a imprimir y pueden indicarse tantos como se quieran separados por comas.
- `sep` establece el separador entre los datos, que por defecto es un espacio en blanco ' '.
- `end` indica la cadena final de la impresión, que por defecto es un cambio de línea `\n`.
- `file` indica la dirección del flujo de salida, que por defecto es la salida estándar `sys.stdout`.

```
1 >>> print('Hola')
2 Hola
3 >>> name = 'Alf'
4 >>> print('Hola', name)
5 Hola Alf
6 >>> print('El valor de pi es', 3.1415)
7 El valor de pi es 3.1415
8 >>> print('Hola', name, sep='')
9 HolaAlf
10 >>> print('Hola', name, end='!\n')
11 Hola Alf!
```

## 3 Estructuras de control

### 3.1 Condicionales (`if`)

```
if condición1:
    bloque código
elif condición2:
    bloque código
...
else:
    bloque código
```

Evalúa la expresión lógica `condición1` y ejecuta el primer bloque de código si es `True`; si no, evalúa las siguientes condiciones hasta llegar a la primera que es `True` y ejecuta el bloque de código asociado. Si ninguna condición es `True` ejecuta el bloque de código después de `else:`.

Pueden aparecer varios bloques `elif` pero solo uno `else` al final.

*Los bloques de código deben estar indentados por 4 espacios.*

La instrucción condicional permite evaluar el estado del programa y tomar decisiones sobre qué código ejecutar en función del mismo.

```
1 >>> edad = 14
2 >>> if edad <= 18 :
3 ...     print('Menor')
4 ... elif edad > 65:
5 ...     print('Jubilado')
6 ... else:
7 ...     print('Activo')
8 ...
9 Menor
10 >>> age = 20
11 >>> if edad <= 18 :
12 ...     print('Menor')
13 ... elif edad > 65:
14 ...     print('Jubilado')
15 ... else:
16 ...     print('Activo')
17 ...
18 Activo
```

### 3.2 Bucles condicionales (while)

**while** condición:  
    bloque código

Repite la ejecución del bloque de código mientras la expresión lógica **condición** sea cierta.

Se puede interrumpir en cualquier momento la ejecución del bloque de código con la instrucción **break**.

*El bloque de código debe estar indentado por 4 espacios.*

```
1 >>> # Pregunta al usuario por un número hasta que introduce 0.
2 >>> num = None
3 >>> while num != 0:
4 ...     num = int(input('Introduce un número: '))
5 ...
6 Introduce un número: 2
7 Introduce un número: 1
8 Introduce un número: 0
9 >>>
```

Alternativa:

```
1 >>> # Pregunta al usuario por un número hasta que introduce 0.
2 >>> while True:
3 ...     num = int(input('Introduce un número: '))
4 ...     if num == 0:
5 ...         break
6 ...
7 Introduce un número: 2
```

```
8 Introduce un número: 1
9 Introduce un número: 0
10 >>>
```

### 3.3 Bucles iterativos (for)

```
for i in secuencia:
    bloque código
```

Repite la ejecución del bloque de código para cada elemento de la secuencia `secuencia`, asignado dicho elemento a `i` en cada repetición.

Se puede interrumpir en cualquier momento la ejecución del bloque de código con la instrucción **break** o saltar la ejecución para un determinado elemento de la secuencia con la instrucción **continue**.

*El bloque de código debe estar indentado por 4 espacios.*

Se utiliza fundamentalmente para recorrer colecciones de objetos como cadenas, listas, tuplas o diccionarios.

A menudo se usan con la instrucción `range`:

- `range(fin)` : Genera una secuencia de números enteros desde 0 hasta `fin-1`.
- `range(inicio, fin, salto)` : Genera una secuencia de números enteros desde `inicio` hasta `fin-1` con un incremento de `salto`.

```
1 >>> palabra = 'Python'
2 >>> for letra in palabra:
3 ...     print(letra)
4 ...
5 P
6 y
7 t
8 h
9 o
10 n
```

```
1 >>> for i in range(1, 10, 2):
2 ...     print(i, end=" ",)
3 ...
4 1, 3, 5, 7, 9, >>>
```

## 4 Tipos de datos estructurados

### 4.1 Listas

Una **lista** es una secuencias ordenadas de objetos de distintos tipos.

Se construyen poniendo los elementos entre corchetes `[ ]` separados por comas.

Se caracterizan por:

- Tienen orden.
- Pueden contener elementos de distintos tipos.
- Son mutables, es decir, pueden alterarse durante la ejecución de un programa.

```
1 # Lista vacía
2 >>> type([])
3 <class 'list'>
4 # Lista con elementos de distintos tipos
5 >>> [1, "dos", True]
6 # Listas anidadas
7 >>> [1, [2, 3], 4]
```

#### 4.1.1 Creación de listas mediante la función `list()`

Otra forma de crear listas es mediante la función `list()`.

- `list(c)` : Crea una lista con los elementos de la secuencia o colección `c`.

Se pueden indicar los elementos separados por comas, mediante una cadena, o mediante una colección de elementos iterable.

```
1 >>> list()
2 []
3 >>> list(1, 2, 3)
4 [1, 2, 3]
5 >>> list("Python")
6 ['P', 'y', 't', 'h', 'o', 'n']
```

#### 4.1.2 Acceso a los elementos de una lista

Se utilizan los mismos operadores de acceso que para cadenas de caracteres.

- `l[i]` : Devuelve el elemento de la lista `l` con el índice `i`.

*El índice del primer elemento de la lista es 0.*

```
1 >>> a = ['P', 'y', 't', 'h', 'o', 'n']
2 >>> a[0]
3 'P'
4 >>> a[5]
5 'n'
6 >>> a[6]
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
```

```
9 IndexError: list index out of range
10 >>> a[-1]
11 'n'
```

### 4.1.3 Sublistas

- `l[i:j:k]` : Devuelve la sublista desde el elemento de `l` con el índice `i` hasta el elemento anterior al índice `j`, tomando elementos cada `k`.

```
1 >>> a = ['P', 'y', 't', 'h', 'o', 'n']
2 >>> a[1:4]
3 ['y', 't', 'h']
4 >>> a[1:1]
5 []
6 >>> a[:-3]
7 ['y', 't', 'h']
8 >>> a[: ]
9 ['P', 'y', 't', 'h', 'o', 'n']
10 >>> a[0:6:2]
11 ['P', 't', 'o']
```

### 4.1.4 Operaciones que no modifican una lista

- `len(l)` : Devuelve el número de elementos de la lista `l`.
- `min(l)` : Devuelve el mínimo elemento de la lista `l` siempre que los datos sean comparables.
- `max(l)` : Devuelve el máximo elemento de la lista `l` siempre que los datos sean comparables.
- `sum(l)` : Devuelve la suma de los elementos de la lista `l`, siempre que los datos se puedan sumar.
- `dato in l` : Devuelve `True` si el dato `dato` pertenece a la lista `l` y `False` en caso contrario.
- `l.index(dato)` : Devuelve la posición que ocupa en la lista `l` el primer elemento con valor `dato`.
- `l.count(dato)` : Devuelve el número de veces que el valor `dato` está contenido en la lista `l`.
- `all(l)` : Devuelve `True` si todos los elementos de la lista `l` son `True` y `False` en caso contrario.
- `any(l)` : Devuelve `True` si algún elemento de la lista `l` es `True` y `False` en caso contrario.

```
1 >>> a = [1, 2, 2, 3]
2 >>> len(a)
3 4
4 >>> min(a)
5 1
6 >>> max(a)
7 3
8 >>> sum(a)
9 8
10 >>> 3 in a
11 True
12 >>> a.index(2)
13 1
```

```
14 >>> a.count(2)
15 2
16 >>> all(a)
17 True
18 >>> any([0, False, 3<2])
19 False
```

#### 4.1.5 Operaciones que modifican una lista

- `l1 + l2` : Crea una nueva lista concatenando los elementos de las listas `l1` y `l2`.
- `l.append(dato)` : Añade `dato` al final de la lista `l`.
- `l.extend(sequencia)` : Añade los datos de `sequencia` al final de la lista `l`.
- `l.insert(índice, dato)` : Inserta `dato` en la posición `índice` de la lista `l` y desplaza los elementos una posición a partir de la posición `índice`.
- `l.remove(dato)` : Elimina el primer elemento con valor `dato` en la lista `l` y desplaza los que están por detrás de él una posición hacia delante.
- `l.pop([índice])` : Devuelve el dato en la posición `índice` y lo elimina de la lista `l`, desplazando los elementos por detrás de él una posición hacia delante.
- `l.sort()` : Ordena los elementos de la lista `l` de acuerdo al orden predefinido, siempre que los elementos sean comparables.
- `l.reverse()` : Invierte el orden de los elementos de la lista `l`.

```
1 >>> a = [1, 3]
2 >>> b = [2, 4, 6]
3 >>> a.append(5)
4 >>> a
5 [1, 3, 5]
6 >>> a.remove(3)
7 >>> a
8 [1, 5]
9 >>> a.insert(1, 3)
10 >>> a
11 [1, 3, 5]
12 >>> b.pop()
13 6
14 >>> c = a + b
15 >>> c
16 [1, 3, 5, 2, 4]
17 >>> c.sort()
18 >>> c
19 [1, 2, 3, 4, 5]
20 >>> c.reverse()
21 >>> c
22 [5, 4, 3, 2, 1]
```

### 4.1.6 Copia de listas

Existen dos formas de copiar listas:

- **Copia por referencia** `l1 = l2`: Asocia la variable `l1` la misma lista que tiene asociada la variable `l2`, es decir, ambas variables apuntan a la misma dirección de memoria. Cualquier cambio que hagamos a través de `l1` o `l2` afectará a la misma lista.
- **Copia por valor** `l1 = list(l2)`: Crea una copia de la lista asociada a `l2` en una dirección de memoria diferente y se la asocia a `l1`. Las variables apuntan a direcciones de memoria diferentes que contienen los mismos datos. Cualquier cambio que hagamos a través de `l1` no afectará a la lista de `l2` y viceversa.

```
1 >>> a = [1, 2, 3]
2 >>> # copia por referencia
3 >>> b = a
4 >>> b
5 [1, 2, 3]
6 >>> b.remove(2)
7 >>> b
8 [1, 3]
9 >>> a
10 [1, 3]
```

```
1 >>> a = [1, 2, 3]
2 >>> # copia por referencia
3 >>> b = list(a)
4 >>> b
5 [1, 2, 3]
6 >>> b.remove(2)
7 >>> b
8 [1, 3]
9 >>> a
10 [1, 2, 3]
```

## 4.2 Tuplas

Una **tupla** es una secuencia ordenada de objetos de distintos tipos.

Se construyen poniendo los elementos entre corchetes ( ) separados por comas.

Se caracterizan por:

- Tienen orden.
- Pueden contener elementos de distintos tipos.
- Son inmutables, es decir, no pueden alterarse durante la ejecución de un programa.

Se usan habitualmente para representar colecciones de datos una determinada estructura semántica, como por ejemplo un vector o una matriz.

```
1 # Tupla vacía
2 type(())
3 <class 'tuple'>
4 # Tupla con elementos de distintos tipos
5 (1, "dos", True)
6 # Vector
7 (1, 2, 3)
8 # Matriz
9 ((1, 2, 3), (4, 5, 6))
```

#### 4.2.1 Creación de tuplas mediante la función `tuple()`

Otra forma de crear tuplas es mediante la función `tuple()`.

- `tuple(c)` : Crea una tupla con los elementos de la secuencia o colección `c`.

Se pueden indicar los elementos separados por comas, mediante una cadena, o mediante una colección de elementos iterable.

```
1 >>> tuple()
2 ()
3 >>> tuple(1, 2, 3)
4 (1, 2, 3)
5 >>> tuple("Python")
6 ('P', 'y', 't', 'h', 'o', 'n')
7 >>> tuple([1, 2, 3])
8 (1, 2, 3)
```

#### 4.2.2 Operaciones con tuplas

El acceso a los elementos de una tupla se realiza del mismo modo que en las listas. También se pueden obtener subtuplas de la misma manera que las sublistas.

Las operaciones de listas que no modifican la lista también son aplicables a las tuplas.

```
1 >>> a = (1, 2, 3)
2 >>> a[1]
3 2
4 >>> len(a)
5 3
6 >>> a.index(3)
7 2
8 >>> 0 in a
9 False
10 >>> b = ((1, 2, 3), (4, 5, 6))
11 >>> b[1]
12 (4, 5, 6)
```



```
13 >>> b[1][2]
14 6
```

### 4.3 Diccionarios

Un diccionario es una colección de pares formados por una *clave* y un *valor* asociado a la clave.

Se construyen poniendo los pares entre llaves { } separados por comas, y separando la clave del valor con dos puntos :.

Se caracterizan por:

- No tienen orden.
- Pueden contener elementos de distintos tipos.
- Son mutables, es decir, pueden alterarse durante la ejecución de un programa.
- Las claves son únicas, es decir, no pueden repetirse en un mismo diccionario, y pueden ser de cualquier tipo de datos inmutable.

```
1 # Diccionario vacío
2 type({})
3 <class 'dict'>
4 # Diccionario con elementos de distintos tipos
5 {'nombre': 'Alfredo', 'despacho': 218, 'email': 'asalber@ceu.es'}
6 # Diccionarios anidados
7 {'nombre_completo': {'nombre': 'Alfredo', 'Apellidos': 'Sánchez Alberca'}}
```

#### 4.3.1 Acceso a los elementos de un diccionario

- `d[clave]` devuelve el valor del diccionario `d` asociado a la clave `clave`. Si en el diccionario no existe esa clave devuelve un error.
- `d.get(clave, valor)` devuelve el valor del diccionario `d` asociado a la clave `clave`. Si en el diccionario no existe esa clave devuelve `valor`, y si no se especifica un valor por defecto devuelve `None`.

```
1 >>> a = {'nombre': 'Alfredo', 'despacho': 218, 'email': 'asalber@ceu.es'}
2 >>> a['nombre']
3 'Alfredo'
4 >>> a['despacho'] = 210
5 >>> a
6 {'nombre': 'Alfredo', 'despacho': 218, 'email': 'asalber@ceu.es'}
7 >>> a.get('email')
8 'asalber@ceu.es'
9 >>> a.get('universidad', 'CEU')
10 'CEU'
```

### 4.3.2 Operaciones que no modifican un diccionario

- `len(d)` : Devuelve el número de elementos del diccionario `d`.
- `min(d)` : Devuelve la mínima clave del diccionario `d` siempre que las claves sean comparables.
- `max(d)` : Devuelve la máxima clave del diccionario `d` siempre que las claves sean comparables.
- `sum(d)` : Devuelve la suma de las claves del diccionario `d`, siempre que las claves se puedan sumar.
- `clave in d` : Devuelve `True` si la clave `clave` pertenece al diccionario `d` y `False` en caso contrario.
- `d.keys()` : Devuelve un iterador sobre las claves de un diccionario.
- `d.values()` : Devuelve un iterador sobre los valores de un diccionario.
- `d.items()` : Devuelve un iterador sobre los pares clave-valor de un diccionario.

```
1 >>> a = {'nombre': 'Alfredo', 'despacho': 218, 'email': 'asalber@ceu.es'}
2 >>> len(a)
3 3
4 >>> min(a)
5 'despacho'
6 >>> 'email' in a
7 True
8 >>> a.keys()
9 dict_keys(['nombre', 'despacho', 'email'])
10 >>> a.values()
11 dict_values([218, 'asalber@ceu.es'])
12 >>> a.items()
13 dict_items([('nombre', 'Alfredo'), ('despacho', 218), ('email', 'asalber@ceu.es')])
```

### 4.3.3 Operaciones que modifican un diccionario

- `d[clave] = valor` : Añade al diccionario `d` el par formado por la clave `clave` y el valor `valor`.
- `d.update(d2)` : Añade los pares del diccionario `d2` al diccionario `d`.
- `d.pop(clave, alternativo)` : Devuelve el valor asociado a la clave `clave` del diccionario `d` y lo elimina del diccionario. Si la clave no está devuelve el valor `alternativo`.
- `d.popitem()` : Devuelve la tupla formada por la clave y el valor del último par añadido al diccionario `d` y lo elimina del diccionario.
- `del d[clave]` : Elimina del diccionario `d` el par con la clave `clave`.
- `d.clear()` : Elimina todos los pares del diccionario `d` de manera que se queda vacío.

```
1 >>> a = {'nombre': 'Alfredo', 'despacho': 218, 'email': 'asalber@ceu.es'}
2 >>> a['universidad'] = 'CEU'
3 >>> a
4 {'nombre': 'Alfredo', 'despacho': 218, 'email': 'asalber@ceu.es', 'universidad': 'CEU'}
5 >>> a.pop('despacho')
6 218
7 >>> a
8 {'nombre': 'Alfredo', 'email': 'asalber@ceu.es', 'universidad': 'CEU'}
```

```
9 >>> a.popitem()
10 ('universidad', 'CEU')
11 >>> a
12 {'nombre': 'Alfredo', 'email': 'asalber@ceu.es'}
13 >>> del a['email']
14 >>> a
15 {'nombre': 'Alfredo'}
16 >>> a.clear()
17 >>> a
18 {}
```

#### 4.3.4 Copia de diccionarios

Existen dos formas de copiar diccionarios:

- **Copia por referencia** `d1 = d2`: Asocia la variable `d1` el mismo diccionario que tiene asociado la variable `d2`, es decir, ambas variables apuntan a la misma dirección de memoria. Cualquier cambio que hagamos a través de `d1` o `d2` afectará al mismo diccionario.
- **Copia por valor** `d1 = dict(d2)`: Crea una copia del diccionario asociado a `d2` en una dirección de memoria diferente y se la asocia a `d1`. Las variables apuntan a direcciones de memoria diferentes que contienen los mismos datos. Cualquier cambio que hagamos a través de `d1` no afectará al diccionario de `d2` y viceversa.

```
1 >>> a = {1:'A', 2:'B', 3:'C'}
2 >>> # copia por referencia
3 >>> b = a
4 >>> b
5 {1:'A', 2:'B', 3:'C'}
6 >>> b.pop(2)
7 >>> b
8 {1:'A', 3:'C'}
9 >>> a
10 {1:'A', 3:'C'}
```

```
1 >>> a = {1:'A', 2:'B', 3:'C'}
2 >>> # copia por referencia
3 >>> b = dict(a)
4 >>> b
5 {1:'A', 2:'B', 3:'C'}
6 >>> b.pop(2)
7 >>> b
8 {1:'A', 3:'C'}
9 >>> a
10 {1:'A', 2:'B', 3:'C'}
```

## 5 Funciones

### 5.1 Funciones (def)

Una función es un bloque de código que tiene asociado un nombre, de manera que cada vez que se quiera ejecutar el bloque de código basta con invocar el nombre de la función.

Para declarar una función se utiliza la siguiente sintaxis:

```
def <nombre-funcion> (<parámetros>):  
    bloque código  
    return <objeto>
```

```
1 >>> def bienvenida():  
2 ...     print('¡Bienvenido a Python!')  
3 ...     return  
4 ...  
5 >>> type(bienvenida)  
6 <class 'function'>  
7 >>> bienvenida()  
8 ¡Bienvenido a Python!
```

#### 5.1.1 Parámetros de una función

Una función puede recibir valores cuando se invoca a través de unas variables conocidas como *parámetros* que se definen entre paréntesis en la declaración de la función. En el cuerpo de la función se pueden usar estos parámetros como si fuesen variables.

```
1 >>> def bienvenida(nombre):  
2 ...     print('¡Bienvenido a Python', nombre + '!')  
3 ...     return  
4 ...  
5 >>> bienvenida('Alf')  
6 ¡Bienvenido a Python Alf!
```

#### 5.1.2 Argumentos de la llamada a una función

Los valores que se pasan a la función en una llamada o invocación concreta de ella se conocen como *argumentos* y se asocian a los parámetros de la declaración de la función.

Los argumentos se pueden indicar de dos formas:

- **Argumentos posicionales:** Se asocian a los parámetros de la función en el mismo orden que aparecen en la definición de la función.

- **Argumentos por nombre:** Se indica explícitamente el nombre del parámetro al que se asocia un argumento de la forma `parametro = argumento`.

```
1 >>> def bienvenida(nombre, apellido):
2 ...     print('¡Bienvenido a Python', nombre, apellido + '!')
3 ...     return
4 ...
5 >>> bienvenida('Alfredo', 'Sánchez')
6 ¡Bienvenido a Python Alfredo Sánchez!
7 >>> bienvenida(apellido = 'Sánchez', nombre = 'Alfredo')
8 ¡Bienvenido a Python Alfredo Sánchez!
```

### 5.1.3 Retorno de una función

Una función puede devolver un objeto de cualquier tipo tras su invocación. Para ello el objeto a devolver debe escribirse detrás de la palabra reservada **return**. Si no se indica ningún objeto, la función no devolverá nada.

```
1 >>> def area_triangulo(base, altura):
2 ...     return base * altura / 2
3 ...
4 >>> area_triangulo(2, 3)
5 3
6 >>> area_triangulo(4, 5)
7 10
```

## 5.2 Argumentos por defecto

En la definición de una función se puede asignar a cada parámetro un argumento por defecto, de manera que si se invoca la función sin proporcionar ningún argumento para ese parámetro, se utiliza el argumento por defecto.

```
1 >>> def bienvenida(nombre, lenguaje = 'Python'):
2 ...     print('¡Bienvenido a', lenguaje, nombre + '!')
3 ...     return
4 ...
5 >>> bienvenida('Alf')
6 ¡Bienvenido a Python Alf!
7 >>> bienvenida('Alf', 'Java')
8 ¡Bienvenido a Java Alf!
```

## 5.3 Pasar un número indeterminado de argumentos

Por último, es posible pasar un número variable de argumentos a un parámetro. Esto se puede hacer de dos formas:

- **\*parametro**: Se antepone un asterisco al nombre del parámetro y en la invocación de la función se pasa el número variable de argumentos separados por comas. Los argumentos se guardan en una lista que se asocia al parámetro.
- **\*\*parametro**: Se anteponen dos asteriscos al nombre del parámetro y en la invocación de la función se pasa el número variable de argumentos por pares **nombre = valor**, separados por comas. Los argumentos se guardan en un diccionario que se asocia al parámetro.

```
1 >>> def menu(*platos):
2 ...     print('Hoy tenemos: ', end='')
3 ...     for plato in platos:
4 ...         print(plato, end=', ')
5 ...     return
6 ...
7 >>> menu('pasta', 'pizza', 'ensalada')
8 Hoy tenemos: pasta, pizza, ensalada,
```

## 5.4 Ámbito de los parámetros y variables de una función

Los parámetros y las variables declaradas dentro de una función son de **ámbito local**, mientras que las definidas fuera de ella son de ámbito **global**.

Tanto los parámetros como las variables del ámbito local de una función sólo están accesibles durante la ejecución de la función, es decir, cuando termina la ejecución de la función estas variables desaparecen y no son accesibles desde fuera de la función.

```
1 >>> def bienvenida(nombre):
2 ...     lenguaje = 'Python'
3 ...     print('¡Bienvenido a', lenguaje, nombre + '!')
4 ...     return
5 ...
6 >>> bienvenida('Alf')
7 ¡Bienvenido a Python Alf!
8 >>> lenguaje
9 Traceback (most recent call last):
10   File "<stdin>", line 1, in <module>
11 NameError: name 'lenguaje' is not defined
```

## 5.5 Ámbito de los parámetros y variables de una función

Si en el ámbito local de una función existe una variable que también existe en el ámbito global, durante la ejecución de la función la variable global queda eclipsada por la variable local y no es accesible hasta que finaliza la ejecución de la función.

```
1 >>> lenguaje = 'Java'
2 >>> def bienvenida():
3 ...     lenguaje = 'Python'
```

```
4 ...     print('¡Bienvenido a', lenguaje + '!')
5 ...     return
6 ...
7 >>> bienvenida()
8 ¡Bienvenido a Python!
9 >>> print(lenguaje)
10 Java
```

## 5.6 Paso de argumentos por referencia

En Python el paso de argumentos a una función es siempre por referencia, es decir, se pasa una referencia al objeto del argumento, de manera que cualquier cambio que se haga dentro de la función mediante el parámetro asociado afectará al objeto original, siempre y cuando este sea mutable.

```
1 >>> primer_curso = ['Matemáticas', 'Física']
2 >>> def añade_asignatura(curso, asignatura):
3 ...     curso.append(asignatura)
4 ...     return
5 ...
6 >>> añade_asignatura(primer_curso, 'Química')
7 >>> print(primer_curso)
8 ['Matemáticas', 'Física', 'Química']
```

## 5.7 Documentación de funciones

Una práctica muy recomendable cuando se define una función es describir lo que la función hace en un comentario.

En Python esto se hace con un **docstring** que es un tipo de comentario especial se hace en la línea siguiente al encabezado de la función entre tres comillas simples `'''` o dobles `"""`.

Después se puede acceder a la documentación de la función con la función `help(<nombre-función>)`.

```
1 >>> def area_triangulo(base, altura):
2 ...     """Función que calcula el área de un triángulo.
3 ...
4 ...     Parámetros:
5 ...         - base: La base del triángulo.
6 ...         - altura: La altura del triángulo.
7 ...     Resultado:
8 ...         El área del triángulo con la base y altura especificadas.
9 ...     """
10 ...     return base * altura / 2
11 ...
12 >>> help(area_triangulo)
13 area_triangulo(base, altura)
14     Función que calcula el área de un triángulo.
15
```

```
16     Parámetros:  
17         - base: La base del triángulo.  
18         - altura: La altura del triángulo.  
19     Resultado:  
20         El área del triángulo con la base y altura especificadas.
```

## 5.8 Funciones recursivas

Una función recursiva es una función que en su cuerpo contiene una llama a si misma.

La recursión es una práctica común en la mayoría de los lenguajes de programación ya que permite resolver las tareas recursivas de manera más natural.

Para garantizar el final de una función recursiva, las sucesivas llamadas tienen que reducir el grado de complejidad del problema, hasta que este pueda resolverse directamente sin necesidad de volver a llamar a la función.

```
1 >>> def factorial(n):  
2 ...     if n == 0:  
3 ...         return 1  
4 ...     else:  
5 ...         return n * factorial(n-1)  
6 ...  
7 >>> f(5)  
8 120
```

### 5.8.1 Funciones recursivas múltiples

Una función recursiva puede invocarse a si misma tantas veces como quiera en su cuerpo.

```
1 >>> def fibonacci(n):  
2 ...     if n <= 1:  
3 ...         return n  
4 ...     else:  
5 ...         return fibonacci(n - 1) + fibonacci(n - 2)  
6 ...  
7 >>> fibonacci(6)  
8 8
```

### 5.8.2 Los riesgos de la recursión

Aunque la recursión permite resolver las tareas recursivas de forma más natural, hay que tener cuidado con ella porque suele consumir bastante memoria, ya que cada llamada a la función crea un nuevo ámbito local con las variables y los parámetros de la función.

En muchos casos es más eficiente resolver la tarea recursiva de forma iterativa usando bucles.



```
1 >>> def fibonacci(n):
2 ...     a, b = 0, 1
3 ...     for i in range(n):
4 ...         a, b = b, a + b
5 ...     return a
6 ...
7 >>> fibonacci(6)
8 8
```

## 5.9 Importación de funciones (import)

Las funciones definidas en un programa o módulo de Python pueden ser importadas y reutilizadas en otros programas.

Existen varias formas de importar módulos y funciones:

- **import M**: Importa el módulo **M** y crea una referencia a él, de manera que pueden invocarse un objeto o función **f** definida en él mediante la sintaxis **M.f**.
- **import M as N**: Importa el módulo **M** y crea una referencia **N** a él, de manera que pueden invocarse un objeto o función **f** definida en él mediante la sintaxis **N.f**. Esta forma es similar a la anterior, pero tiene se suele usar cuando el nombre del módulo es muy largo para utilizar un alias más corto.
- **from M import \***: Importa el módulo **M** y crea referencias a todos los objetos públicos (aquellos que no empiezan por el carácter **\_**) definidos en el módulo. De esta manera para invocar un objeto del módulo no hace falta precederlo por el nombre del módulo, basta con escribir su nombre.
- **from M import f, g, ...**: Importa el módulo **M** y crea referencias a los objetos **f**, **g**, ..., de manera que pueden ser invocados por su nombre.

```
1 >>> import calendar
2 >>> print(calendar.month(2019, 4))
3 April 2019
4 Mo Tu We Th Fr Sa Su
5  1  2  3  4  5  6  7
6  8  9 10 11 12 13 14
7 15 16 17 18 19 20 21
8 22 23 24 25 26 27 28
9 29 30
```

```
1 >>> from math import *
2 >>> cos(pi)
3 -1.0
```

### 5.9.1 Módulos de la librería estándar más importantes

No necesitan instalarse porque vienen incluidos en la distribución de Python.

- `sys`: Funciones y parámetros específicos del sistema operativo.
- `os`: Interfaz con el sistema operativo.
- `os.path`: Funciones de acceso a las rutas del sistema.
- `io`: Funciones para manejo de flujos de datos y ficheros.
- `string`: Funciones con cadenas de caracteres.
- `datetime`: Funciones para fechas y tiempos.
- `math`: Funciones y constantes matemáticas.
- `statistics`: Funciones estadísticas.
- `random`: Generación de números pseudo-aleatorios.

### 5.9.2 Otros módulos imprescindibles

Necesitan instalarse.

- `NumPy`: Funciones matemáticas avanzadas y arrays.
- `SciPy`: Más funciones matemáticas para aplicaciones científicas.
- `matplotlib`: Análisis y representación gráfica de datos.
- `Pandas`: Funciones para el manejo y análisis de estructuras de datos.
- `Request`: Acceso a internet por http.

## 5.10 Programación funcional

En Python las funciones son objetos de primera clase, es decir, que pueden pasarse como argumentos de una función, al igual que el resto de los tipos de datos.

```
1 >>> def aplica(funcion, argumento):
2 ...     return funcion(argumento)
3 ...
4 >>> def cuadrado(n):
5 ...     return n*n
6 ...
7 >>> def cubo(n):
8 ...     return n**3
9 ...
10 >>> aplica(cuadrado, 5)
11 25
12 >>> aplica(cubo, 5)
13 125
```

### 5.10.1 Funciones anónimas (lambda)

Existe un tipo especial de funciones que no tienen nombre asociado y se conocen como **funciones anónimas** o **funciones lambda**.

La sintaxis para definir una función anónima es

```
lambda <parámetros> : <expresión>
```

Estas funciones se suelen asociar a una variable o parámetro desde la que hacer la llamada.

```
1 >>> area = lambda base, altura : base * altura
2 >>> area(4, 5)
3 10
```

### 5.10.2 Aplicar una función a todos los elementos de una colección iterable (map)

`map(f, c)` : Devuelve un objeto iterable con los resultados de aplicar la función `f` a los elementos de la colección `c`. Si la función `f` requiere `n` argumentos entonces deben pasarse `n` colecciones con los argumentos. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

```
1 >>> def cuadrado(n):
2 ...     return n * n
3 ...
4 >>> list(map(cuadrado, [1, 2, 3]))
5 [1, 4, 9]
```

```
1 >>> def rectangulo(a, b):
2 ...     return a * b
3 ...
4 >>> tuple(map(rectangulo, (1, 2, 3), (4, 5, 6)))
5 (4, 10, 18)
```

### 5.10.3 Filtrar los elementos de una colección iterable (filter)

`filter(f, c)` : Devuelve un objeto iterable con los elementos de la colección `c` que devuelven `True` al aplicarles la función `f`. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

`f` debe ser una función que recibe un argumento y devuelve un valor booleano.

```
1 >>> def par(n):
2 ...     return n % 2 == 0
3 ...
4 >>> list(filter(par, range(10)))
5 [0, 2, 4, 6, 8]
```

### 5.10.4 Combinar los elementos de varias colecciones iterables (zip)

`zip(c1, c2, ...)` : Devuelve un objeto iterable cuyos elementos son tuplas formadas por los elementos que ocupan la misma posición en las colecciones `c1`, `c2`, etc. El número de elementos de las tuplas es el nú-

mero de colecciones que se pasen. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

```
1 >>> asignaturas = ['Matemáticas', 'Física', 'Química', 'Economía']
2 >>> notas = [6.0, 3.5, 7.5, 8.0]
3 >>> list(zip(asignaturas, notas))
4 [('Matemáticas', 6.0), ('Física', 3.5), ('Química', 7.5), ('Economía',
5    8.0)]
6 >>> dict(zip(asignaturas, notas[:3]))
7 {'Matemáticas': 6.0, 'Física': 3.5, 'Química': 7.5}
```

### 5.10.5 Operar todos los elementos de una colección iterable (`reduce`)

`reduce(f, l)` : Aplicar la función `f` a los dos primeros elementos de la secuencia `l`. Con el valor obtenido vuelve a aplicar la función `f` a ese valor y el siguiente de la secuencia, y así hasta que no quedan más elementos en la lista. Devuelve el valor resultado de la última aplicación de la función `f`.

La función `reduce` está definida en el módulo `functools`.

```
1 >>> from functools import reduce
2 >>> def producto(n, m):
3 ...     return n * m
4 ...
5 >>> reduce(producto, range(1, 5))
6 24
```

## 5.11 Comprensión de colecciones

En muchas aplicaciones es habitual aplicar una función o realizar una operación con los elementos de una colección (lista, tupla o diccionario) y obtener una nueva colección de elementos transformados. Aunque esto se puede hacer recorriendo la secuencia con un bucle iterativo, y en programación funcional mediante la función `map`, Python incorpora un mecanismo muy potente que permite esto mismo de manera más simple.

### 5.11.1 Comprensión de listas

[*expresion* **for** *variable* **in** *lista* **if** *condicion*]

Esta instrucción genera la lista cuyos elementos son el resultado de evaluar la expresión *expresion*, para cada valor que toma la variable *variable*, donde *variable* toma todos los valores de la lista *lista* que cumplen la condición *condición*.

```
1 >>> [x ** 2 for x in range(10)]
2 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
3 >>> [x for x in range(10) if x % 2 == 0]
4 [0, 2, 4, 6, 8]
```

```
5 >>> [x ** 2 for x in range(10) if x % 2 == 0]
6 [0, 4, 16, 36, 64]
7 >>> notas = {'Carmen':5, 'Antonio':4, 'Juan':8, 'Mónica':9, 'María': 6,
8             'Pablo':3}
9 >>> [nombre for (nombre, nota) in notas.items() if nota >= 5]
10 ['Carmen', 'Juan', 'Mónica', 'María']
```

### 5.11.2 Comprensión de diccionarios

`{expresion-clave:expresion-valor for variables in lista if condicion}`

Esta instrucción genera el diccionario formado por los pares cuyas claves son el resultado de evaluar la expresión *expresion-clave* y cuyos valores son el resultado de evaluar la expresión *expresion-valor*, para cada valor que toma la variable *variable*, donde *variable* toma todos los valores de la lista *lista* que cumplen la condición *condición*.

```
1 >>> {palabra:len(palabra) for palabra in ['I', 'love', 'Python']}
2 {'I': 1, 'love': 4, 'Python': 6}
3 >>> notas = {'Carmen':5, 'Antonio':4, 'Juan':8, 'Mónica':9, 'María': 6,
4             'Pablo':3}
5 >>> {nombre: nota +1 for (nombre, nota) in notas.items() if nota >= 5})
6 {'Carmen': 6, 'Juan': 9, 'Mónica': 10, 'María': 7}
```

## 6 Ficheros y excepciones

### 6.1 Ficheros

Hasta ahora hemos visto como interactuar con un programa a través del teclado (entrada de datos) y la terminal (salida), pero en la mayor parte de las aplicaciones reales tendremos que leer y escribir datos en ficheros.

Al utilizar ficheros para guardar los datos estos perdurarán tras la ejecución del programa, pudiendo ser consultados o utilizados más tarde.

Las operaciones más habituales con ficheros son:

- Crear un fichero.
- Escribir datos en un fichero.
- Leer datos de un fichero.
- Borrar un fichero.

#### 6.1.1 Creación y escritura de ficheros

Para crear un fichero nuevo se utiliza la instrucción

`open(ruta, 'w')` : Crea el fichero con la ruta `ruta`, lo abre en modo escritura (el argumento 'w' significa *write*) y devuelve un objeto que lo referencia.

Si el fichero indicado por la ruta ya existe en el sistema, se reemplazará por el nuevo.

Una vez creado el fichero, para escribir datos en él se utiliza el método

`fichero.write(c)` : Escribe la cadena `c` en el fichero referenciado por `fichero`.

```
1 >>> f = open('bienvenida.txt', 'w')
2 ... f.write('¡Bienvenido a Python!')
```

### 6.1.2 Añadir datos a un fichero

Si en lugar de crear un fichero nuevo queremos añadir datos a un fichero existente se debe utilizar la instrucción

`open(ruta, 'a')` : Abre el fichero con la ruta `ruta` en modo añadir (el argumento 'a' significa *append*) y devuelve un objeto que lo referencia.

Una vez abierto el fichero, se utiliza el método de escritura anterior y los datos se añaden al final del fichero.

```
1 >>> f = open('bienvenida.txt', 'a')
2 ... f.write('\n¡Hasta pronto!')
```

### 6.1.3 Leer datos de un fichero

Para abrir un fichero en modo lectura se utiliza la instrucción

`open(ruta, 'r')` : Abre el fichero con la ruta `ruta` en modo lectura (el argumento 'r' significa *read*) y devuelve un objeto que lo referencia.

Una vez abierto el fichero, se puede leer todo el contenido del fichero o se puede leer línea a línea.

### 6.1.4 Leer datos de un fichero

`fichero.read()` : Devuelve todos los datos contenidos en `fichero` como una cadena de caracteres.

`fichero.readlines()` : Devuelve una lista de cadenas de caracteres donde cada cadena es una línea del fichero referenciado por `fichero`.

```
1 >>> f = open('bienvenida.txt', 'r')
2 ... print(f.read())
3 ¡Bienvenido a Python!
4 ¡Hasta pronto!
```

```
1 >>> f = open('bienvenida.txt', 'r')
2 ... lineas = print(f.readlines())
3 >>> print(lineas)
4 ['Bienvenido a Python!\n', '¡Hasta pronto!']
```

### 6.1.5 Cerrar un fichero

Para cerrar un fichero se utiliza el método

`fichero.close()` : Cierra el fichero referenciado por el objeto `fichero`.

Cuando se termina de trabajar con un fichero conviene cerrarlo, sobre todo si se abre en modo escritura, ya que mientras está abierto en este modo no se puede abrir por otra aplicación. Si no se cierra explícitamente un fichero, Python intentará cerrarlo cuando estime que ya no se va a usar más.

```
1 >>> f = open('bienvenida.txt'):
2 ... print(f.read())
3 ... f.close() # Cierre del fichero
4 ...
5 ¡Bienvenido a Python!
6 ¡Hasta pronto!
```

### 6.1.6 Renombrado y borrado de un fichero

Para renombrar o borrar un fichero se utilizan funciones del módulo `os`.

`os.rename(ruta1, ruta2)` : Renombra y mueve el fichero de la ruta `ruta1` a la ruta `ruta2`.

`os.remove(ruta)` : Borra el fichero de la ruta `ruta`.

Antes de borrar o renombrar un directorio conviene comprobar que existe para que no se produzca un error. Para ello se utiliza la función

`os.path.isfile(ruta)` : Devuelve `True` si existe un fichero en la ruta `ruta` y `False` en caso contrario.

### 6.1.7 Renombrado y borrado de un fichero o directorio

```
1 >>> import os
2 >>> f = 'bienvenida.txt'
3 >>> if os.path.isfile(f):
4 ...     os.rename(f, 'saludo.txt') # renombrado
5 ... else:
6 ...     print('¡El fichero', f, 'no existe!')
7 ...
8 >>> f = 'saludo.txt'
9 >>> if os.path.isfile(f):
10 ...     os.remove(f) # borrado
```

```
11 ... else:
12 ...     print('¡El fichero', f, 'no existe!')
13 ...
```

### 6.1.8 Creación, cambio y eliminación de directorios

Para trabajar con directorios también se utilizan funciones del módulo `os`.

`os.mkdir(ruta)` : Crea un nuevo directorio en la ruta `ruta`.

`os.chdir(ruta)` : Cambia el directorio actual al indicado por la ruta `ruta`.

`os.getcwd()` : Devuelve una cadena con la ruta del directorio actual.

`os.rmdir(ruta)` : Borra el directorio de la ruta `ruta`, siempre y cuando esté vacío.

### 6.1.9 Leer un fichero de internet

Para leer un fichero de internet hay que utilizar la función `urlopen` del módulo `urllib.request`.

`urlopen(url)` : Abre el fichero con la `url` especificada y devuelve un objeto del tipo fichero al que se puede acceder con los métodos de lectura de ficheros anteriores.

```
1 >>> from urllib import request
2 >>> f = request.urlopen('https://raw.githubusercontent.com/asalber/
   asalber.github.io/master/README.md')
3 >>> datos = f.read()
4 >>> print(datos.decode('utf-8'))
5 Aprende con Alf
6 =====
7
8 Este es el repositorio del sitio web Aprende con Alf: http://
   aprendeconalf.es
```

## 6.2 Control de errores mediante excepciones

Python utiliza un objeto especial llamado **excepción** para controlar cualquier error que pueda ocurrir durante la ejecución de un programa.

Cuando ocurre un error durante la ejecución de un programa, Python crea una excepción. Si no se controla esta excepción la ejecución del programa se detiene y se muestra el error (*traceback*).

```
1 >>> print(1 / 0) # Error al intentar dividir por 0.
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   ZeroDivisionError: division by zero
```



### 6.2.1 Tipos de excepciones

Los principales excepciones definidas en Python son:

- `TypeError` : Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.
- `ZeroDivisionError` : Ocurre cuando se intenta dividir por cero.
- `OverflowError` : Ocurre cuando un cálculo excede el límite para un tipo de dato numérico.
- `IndexError` : Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.
- `KeyError` : Ocurre cuando se intenta acceder a un diccionario con una clave que no existe.
- `FileNotFoundError` : Ocurre cuando se intenta acceder a un fichero que no existe en la ruta indicada.
- `ImportError` : Ocurre cuando falla la importación de un módulo.

Consultar la documentación de Python para ver la [lista de excepciones predefinidas](#).

### 6.2.2 Control de excepciones

**try – except – else** Para evitar la interrupción de la ejecución del programa cuando se produce un error, es posible controlar la excepción que se genera con la siguiente instrucción:

```
try:
    bloque código 1
except excepción:
    bloque código 2
else:
    bloque código 3
```

Esta instrucción ejecuta el primer bloque de código y si se produce un error que genera una excepción del tipo *excepción* entonces ejecuta el segundo bloque de código, mientras que si no se produce ningún error, se ejecuta el tercer bloque de código.

### 6.2.3 Control de excepciones

```
1 >>> def division(a, b):
2 ...     try:
3 ...         result = a / b
4 ...     except ZeroDivisionError:
5 ...         print('¡No se puede dividir por cero!')
6 ...     else:
7 ...         print(result)
8 ...
9 >>> division(1, 0)
10 ¡No se puede dividir por cero!
11 >>> division(1, 2)
12 0.5
```

```
1 >>> try:
2 ...     f = open('fichero.txt') # El fichero no existe
3 ... except FileNotFoundError:
4 ...     print('¡El fichero no existe!')
5 ... else:
6 ...     print(f.read())
7 ¡El fichero no existe!
```

## 7 La librería Numpy

### 7.1 La librería NumPy

NumPy es una librería de Python especializada en el cálculo numérico y el análisis de datos, especialmente para un gran volumen de datos.

Incorpora una nueva clase de objetos llamados **arrays** que permite representar colecciones de datos de un mismo tipo en varias dimensiones, y funciones muy eficientes para su manipulación.



Figura 1: Logo librería numpy

### 7.2 La clase de objetos array

Un array es una estructura de datos de un mismo tipo organizada en forma de tabla o cuadrícula de distintas dimensiones.

Las dimensiones de un array también se conocen como **ejes**.

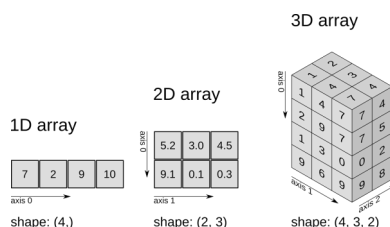


Figura 2: Arrays

### 7.3 Creación de arrays

Para crear un array se utiliza la siguiente función de NumPy

`np.array(secuencia)` : Crea un array a partir de la lista o tupla `lista` y devuelve una referencia a él. El número de dimensiones del array dependerá de las listas o tuplas anidadas en `lista`:

- Para una lista de valores se crea un array de una dimensión, también conocido como **vector**.
- Para una lista de listas de valores se crea un array de dos dimensiones, también conocido como **matriz**.
- Para una lista de listas de listas de valores se crea un array de tres dimensiones, también conocido como **cubo**.
- Y así sucesivamente. No hay límite en el número de dimensiones del array más allá de la memoria disponible en el sistema.

Los elementos de la lista o tupla deben ser del mismo tipo.

```
1 >>> # Array de una dimensión
2 >>> a1 = np.array([1, 2, 3])
3 >>> print(a1)
4 [1 2 3]
5 >>> # Array de dos dimensiones
6 >>> a2 = np.array([[1, 2, 3], [4, 5, 6]])
7 >>> print(a2)
8 [[1 2 3]
9  [4 5 6]]
10 >>> # Array de tres dimensiones
11 >>> a3 = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
12 >>> print(a3)
13 [[[ 1  2  3]
14   [ 4  5  6]]
15
16  [[ 7  8  9]
17   [10 11 12]]]
```

Otras funciones útiles que permiten generar arrays son:

`np.empty(dimensiones)` : Crea y devuelve una referencia a un array vacío con las dimensiones especificadas en la tupla `dimensiones`.

`np.zeros(dimensiones)` : Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla `dimensiones` cuyos elementos son todos ceros.

`np.ones(dimensiones)` : Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla `dimensiones` cuyos elementos son todos unos.

`np.full(dimensiones, valor)` : Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla `dimensiones` cuyos elementos son todos `valor`.

`np.identity(n)` : Crea y devuelve una referencia a la matriz identidad de dimensión `n`.

`np.arange(inicio, fin, salto)` : Crea y devuelve una referencia a un array de una dimensión cuyos elementos son la secuencia desde `inicio` hasta `fin` tomando valores cada `salto`.

`np.linspace(inicio, fin, n)` : Crea y devuelve una referencia a un array de una dimensión cuyos elementos son la secuencia de `n` valores equidistantes desde `inicio` hasta `fin`.

`np.random.random(dimENSIONES)` : Crea y devuelve una referencia a un array con las dimensiones especificadas en la tupla `dimensiones` cuyos elementos son aleatorios.

```
1 >>> print(np.zeros(3,2))
2 [[0. 0. 0.]
3  [0. 0. 0.]]
4 >>> print(np.identity(3))
5 [[1. 0. 0.]
6  [0. 1. 0.]
7  [0. 0. 1.]]
8 >>> print(np.arange(1, 10, 2))
9 [1 3 5 7 9]
10 >>> print(np.linspace(0, 10, 5))
11 [ 0.   2.5  5.   7.5 10.]
```

### 7.3.1 Atributos de un array

Existen varios atributos y funciones que describen las características de un array.

`a.ndim` : Devuelve el número de dimensiones del array `a`.

`a.shape` : Devuelve una tupla con las dimensiones del array `a`.

`a.size` : Devuelve el número de elementos del array `a`.

`a.dtype` : Devuelve el tipo de datos de los elementos del array `a`.

### 7.3.2 Acceso a los elementos de un array

Para acceder a los elementos contenidos en un array se usan índices al igual que para acceder a los elementos de una lista, pero indicando los índices de cada dimensión separados por comas.

Al igual que para listas, los índices de cada dimensión comienzan en 0.

También es posible obtener subarrays con el operador dos puntos : indicando el índice inicial y el siguiente al final para cada dimensión, de nuevo separados por comas.

```
1 >>> a = np.array([[1, 2, 3], [4, 5, 6]])
2 >>> print(a[1, 0]) # Acceso al elemento de la fila 1 columna 0
3 4
4 >>> print(a[1][0]) # Otra forma de acceder al mismo elemento
5 4
6 >>> print(a[:, 0:2])
7 [[1 2]
8  [4 5]]
```

### 7.3.3 Filtrado de elementos de un array

Una característica muy útil de los arrays es que es muy fácil obtener otro array con los elementos que cumplen una condición.

`a[condición]` : Devuelve una lista con los elementos del array `a` que cumplen la condición `condición`.

```
1 >>> a = np.array([[1, 2, 3], [4, 5, 6]])
2 >>> print(a[(a % 2 == 0)])
3 [2 4 6]
4 >>> print(a[(a % 2 == 0) & (a > 2)])
5 [2 4]
```

### 7.3.4 Operaciones matemáticas con arrays

Existen dos formas de realizar operaciones matemáticas con arrays: a nivel de elemento y a nivel de array.

Las operaciones a nivel de elemento operan los elementos que ocupan la misma posición en dos arrays. Se necesitan, por tanto, dos arrays con las mismas dimensiones y el resultado es una array de la misma dimensión.

Los operadores matemáticos `+`, `-`, `*`, `/`, `%`, `**` se utilizan para la realizar suma, resta, producto, cociente, resto y potencia a nivel de elemento.

```
1 >>> a = np.array([[1, 2, 3], [4, 5, 6]])
2 >>> b = np.array([[1, 1, 1], [2, 2, 2]])
3 >>> print(a + b)
4 [[2 3 4]
5  [6 7 8]]
6 >>> print(a / b)
7 [[1.  2.  3. ]
8  [2.  2.5 3. ]]
9 >>> print(a ** 2)
10 [[ 1  4  9]
11  [16 25 36]]
```

### 7.3.5 Operaciones matemáticas a nivel de array

Para realizar el producto matricial se utiliza el método

`a.dot(b)` : Devuelve el array resultado del producto matricial de los arrays `a` y `b` siempre y cuando sus dimensiones sean compatibles.

Y para trasponer una matriz se utiliza el método

`a.T` : Devuelve el array resultado de trasponer el array `a`.

```
1 >>> a = np.array([[1, 2, 3], [4, 5, 6]])
2 >>> b = np.array([[1, 1], [2, 2], [3, 3]])
3 >>> print(a.dot(b))
```

```

4  [[14 14]
5   [32 32]]
6  >>> print(a.T)
7  [[1 4]
8   [2 5]
9   [3 6]]

```

## 8 La librería Pandas

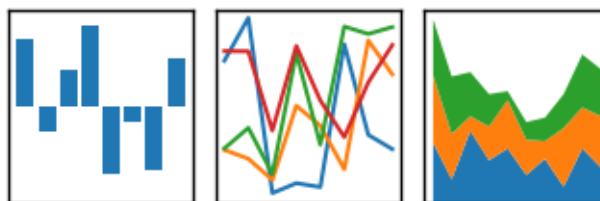
### 8.1 La librería Pandas

**Pandas** es una librería de Python especializada en el manejo y análisis de estructuras de datos.

Las principales características de esta librería son:

- Define el tipo de objetos **DataFrame** para manejar conjuntos de datos.
- Permite leer y escribir fácilmente ficheros en formato CSV, Excel y bases de datos SQL.
- Ofrece métodos para reordenar, dividir y combinar conjuntos de datos.
- Permite trabajar con series temporales.
- Ofrece un rendimiento muy eficiente.

**pandas**  
 $y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$



**Figura 3:** Logo librería Pandas

#### 8.1.1 La clase de objetos DataFrame

Un objeto del tipo **DataFrame** define un conjunto de datos estructurado en forma de tabla donde las columnas suelen contener datos de variables, es decir, todos los datos de una misma columna son del mismo tipo, y las filas de individuos que pueden contener datos de distintos tipos.

Nombre	Edad	Grado	Correo
'María'	18	'Economía'	'maria@gmail.com'
'Luis'	22	'Medicina'	'luis@yahoo.es'
'Carmen'	20	'Arquitectura'	'carmen@gmail.com'

Nombre	Edad	Grado	Correo
'Antonio'	21	'Economía'	'antonio@gmail.com'

### 8.1.2 Creación de un DataFrame a partir de un diccionario

`DataFrame(diccionario)` : Crea un objeto del tipo `DataFrame` con los datos del diccionario `diccionario`. Las claves del diccionario serán los nombres de las columnas, mientras que los valores asociados a las claves serán listas con los valores de las columnas.

```

1 >>> import pandas as pd
2 >>> datos = {'nombre': ['María', 'Luis', 'Carmen', 'Antonio'],
3 ... 'edad': [18, 22, 20, 21],
4 ... 'grado': ['Economía', 'Medicina', 'Arquitectura', 'Economía'],
5 ... 'correo': ['maria@gmail.com', 'luis@yahoo.es', 'carmen@gmail.com', 'antonio@gmail.com']}
6 ... }
7 >>> df = pd.DataFrame(datos)
8 >>> print(df)
9      nombre  edad      grado      correo
10 0   María   18   Economía  maria@gmail.com
11 1    Luis   22   Medicina   luis@yahoo.es
12 2   Carmen  20  Arquitectura  carmen@gmail.com
13 3  Antonio  21    Economía  antonio@gmail.com

```

### 8.1.3 Importación de ficheros

Dependiendo del tipo de fichero, existen distintas funciones para importar un `DataFrame` desde un fichero.

`read_csv(fichero.csv, delimitador)` : Devuelve un objeto del tipo `DataFrame` con los datos del fichero CSV `fichero.csv` usando como separador de los datos la cadena `delimitador`.

`read_excel(fichero.xlsx, hoja)` : Devuelve un objeto del tipo `DataFrame` con los datos de la hoja de cálculo `hoja` del fichero Excel `fichero.xlsx`.

```

1 >>> import pandas as pd
2 >>> # Importación del fichero datos-colesterol.csv
3 >>> df = pd.read_csv(
4 'https://raw.githubusercontent.com/asalber/manual-python/master/datos/colesterol.csv')
5 >>> print(df.head())
6
7      nombre  edad  sexo  peso  altura
8 0  José Luis Martínez Izquierdo   18   H   85   179
9    182
10 1      Rosa Díaz Díaz   32   M   65   173
11    232

```

9	2	Javier García Sánchez	24	H	71	181
	191					
10	3	Carmen López Pinzón	35	M	65	170
	200					
11	4	Marisa López Collado	46	M	51	158
	148					

#### 8.1.4 Exportación de ficheros

También existen funciones para exportar un DataFrame a un fichero con diferentes formatos.

`df.to_csv(fichero.csv, delimitador)` : Exporta el DataFrame `df` al fichero `fichero.csv` en formato CSV usando como separador de los datos la cadena `delimitador`.

`df.to_excel(fichero.xlsx, sheet_name = hoja)` : Exporta el DataFrame `df` a la hoja de cálculo `hoja` del fichero `fichero.xlsx` en formato Excel.

#### 8.1.5 Atributos de un DataFrame

Existen varias propiedades o métodos para ver las características de un DataFrame.

`df.info()` : Devuelve información (número de filas, número de columnas, nombre y tipo de las columnas y memoria usado) sobre el DataFrame `df`.

`df.shape` : Devuelve una tupla con el número de filas y columnas del DataFrame `df`.

`df.columns` : Devuelve el nombre de las columnas del DataFrame `df`.

`df.head()` : Devuelve las 5 primeras filas del DataFrame `df`.

`df.tail()` : Devuelve las 5 últimas filas del DataFrame `df`.

#### 8.1.6 Acceso a los elementos de un DataFrame mediante índices

El acceso a los datos de un DataFrame se puede hacer a través de índices o través de los nombres de las filas y columnas. Para acceder a los elementos de un DataFrame mediante índices se utiliza un método similar al de las listas o arrays.

`df.iloc[i, j]` : Devuelve el elemento que se encuentra en la fila `i` y la columna `j` del DataFrame `df`. Puede indicarse secuencias de índices para obtener partes del DataFrame.

`df.iloc[i]` : Devuelve la fila `i` del DataFrame `df`.

```
1 >>> import pandas as pd
2 >>> df = pd.read_csv(
3   'https://raw.githubusercontent.com/asalber/manual-python/master/datos/
   colesterol.csv')
4 >>> print(df.iloc[1, 3])
```



```
5 65
6 >>> print(df.iloc[1, :2])
7 nombre      Rosa Díaz Díaz
8 edad                32
```

### 8.1.7 Acceso a los elementos de un DataFrame mediante nombres

`df.loc[fila, columna]` : Devuelve el elemento que se encuentra en la fila de nombre *fila* y la columna de nombre *columna* del DataFrame *df*.

`df[columna]` : Devuelve los elementos de la columna de nombre *columna* del DataFrame *df*.

```
1 >>> import pandas as pd
2 >>> df = pd.read_csv(
3     'https://raw.githubusercontent.com/asalber/manual-python/master/datos/
4     colesterol.csv')
5 >>> print(df.loc[2, 'colesterol'])
6 191
7 >>> print(df.loc[:3, ('colesterol', 'peso')])
8     colesterol  peso
9 1          232    65
10 2          191    71
11 3          200    65
12 >>> print(df['colesterol'])
13 0      182
14 1      232
15 2      191
16 3      200
17 ...
```

### 8.1.8 Operaciones sobre columnas

Puesto que los datos de una misma columna de un DataFrame tienen que ser del mismo tipo, es fácil aplicar la misma operación a todos los elementos de la columna.

```
1 >>> import pandas as pd
2 >>> df = pd.read_csv(
3     'https://raw.githubusercontent.com/asalber/manual-python/master/datos/
4     colesterol.csv')
5 >>> print(df['altura']/100)
6 0      1.79
7 1      1.73
8 2      1.81
9 ...
10 >>> print(df['sexo']=='M')
11 0      False
12 1       True
```

```
13 2      False
14 ...
```

### 8.1.9 Aplicar funciones a columnas

Para aplicar funciones a todos los elementos de una columna se utiliza el método

`df[columna].apply(f)` : Devuelve una secuencia con los valores que resulta de aplicar la función `f` a los elementos de la columna de nombre `columna` del DataFrame `df`.

```
1 >>> import pandas as pd
2 >>> from math import log
3 >>> df = pd.read_csv(
4   'https://raw.githubusercontent.com/asalber/manual-python/master/datos/
   colesterol.csv')
5 >>> print(df['altura'].apply(log))
6 0      5.187386
7 1      5.153292
8 2      5.198497
9 ...
```

### 8.1.10 Filtrado de un DataFrame

Una operación bastante común con un DataFrame es obtener las filas que cumplen una determinada condición.

`df[condicion]` : Devuelve un DataFrame con las filas del DataFrame `df` que se corresponden con el valor `True` de la secuencia booleana `condicion`. `condicion` debe ser una secuencia de valores booleanos de la misma longitud que el número de filas del DataFrame.

```
1 >>> import pandas as pd
2 >>> df = pd.read_csv(
3   'https://raw.githubusercontent.com/asalber/manual-python/master/datos/
   colesterol.csv')
4 >>> print(df[(df['sexo']=='H') & (df['colesterol'] > 260)])
5           nombre  edad sexo  peso  altura  colesterol
6 6  Antonio Fernández Ocaña    51    H    62    172      276
7 9  Santiago Reillo Manzano    46    H    75    185      280
```

## 8.2 Eliminar filas y columnas de un DataFrame

Para eliminar filas y columnas de un DataFrame se utiliza el método

`df.drop(filas)` : Devuelve el DataFrame que resulta de eliminar las filas con los nombres indicados en la secuencia `filas` del DataFrame `df`.

`df.drop(columnas, axis=1)`: Devuelve el DataFrame que resulta de eliminar las columnas con los nombres indicados en la secuencia `columnas` del DataFrame `df`.

```

1 >>> import pandas as pd
2 >>> df = pd.read_csv(
3   'https://raw.githubusercontent.com/asalber/manual-python/master/datos/
   colesterol.csv')
4 >>> print(df.drop([1, 3]))
5 0      José Luis Martínez Izquierdo    18    H    85    179
   182
6 2      Javier García Sánchez    24    H    71    181
   191
7 4      Marisa López Collado    46    M    51    158
   148
8 ...
9 >>> print(df.drop(['peso', 'altura'], axis=1))
10 0      José Luis Martínez Izquierdo    18    H    182
11 1      Rosa Díaz Díaz    32    M    232
12 2      Javier García Sánchez    24    H    191
13 ...

```

### 8.2.1 Ordenar un DataFrame

Para ordenar un DataFrame de acuerdo a los valores de una determinada columna se utiliza el método

`df.sort_values(columna)`: Devuelve el DataFrame que resulta de ordenar ascendentemente el DataFrame `df` según los valores de la columna `columna`.

`df.sort_values(columna, ascending=False)`: Devuelve el DataFrame que resulta de ordenar descendientemente el DataFrame `df` según los valores de la columna `columna`.

```

1 >>> import pandas as pd
2 >>> df = pd.read_csv(
3   'https://raw.githubusercontent.com/asalber/manual-python/master/datos/
   colesterol.csv')
4 print(df.sort_values('colesterol'))
5      nombre  edad  sexo  peso  altura
6 4      Marisa López Collado    46    M    51    158
   148
7 0      José Luis Martínez Izquierdo    18    H    85    179
   182
8 2      Javier García Sánchez    24    H    71    181
   191
9 13      Carolina Rubio Moreno    20    M    61    177
   194
10 ...

```

### 8.2.2 Reestructurar un DataFrame: Convertir columnas en filas

A menudo la disposición de los datos en un DataFrame no es la adecuada para su tratamiento y es necesario reestructurar el DataFrame.

Para convertir columnas en filas se utiliza el método

```
df.melt(id_vars, var_name, var_value)
```

```
1 >>> import pandas as pd
2 >>> datos = {'nombre':['María', 'Luis', 'Carmen'],
3 ... 'edad':[18, 22, 20],
4 ... 'Matemáticas':[8.5, 7, 3.5],
5 ... 'Física':[8, 6.5, 5],
6 ... 'Química':[6.5, 4, 9]}
7 >>> df = pd.DataFrame(datos)
8 >>> df1 = df.melt(id_vars=['nombre', 'edad'], var_name='asignatura',
9 ... value_name='nota')
9 >>> print(df1)
10  nombre  edad  asignatura  nota
11 0   María   18  Matemáticas  8.5
12 1    Luis   22  Matemáticas  7.0
13 2  Carmen   20  Matemáticas  3.5
14 3   María   18    Física    8.0
15 4    Luis   22    Física    6.5
16 5  Carmen   20    Física    5.0
17 6   María   18    Química    6.5
18 7    Luis   22    Química    4.0
19 8  Carmen   20    Química    9.0
```

### 8.2.3 Reestructurar un DataFrame: Convertir filas en columnas

Para convertir columnas en filas se utiliza el método

```
df.pivot(index, columns, values)
```

```
1 # Continuación del código anterior
2 >>> print(df1.pivot(index='nombre', columns='asignatura', values='nota'
3 ... ))
4 asignatura  Física  Matemáticas  Química
5 nombre
6 Carmen      5.0      3.5      9.0
7 Luis        6.5      7.0      4.0
8 María       8.0      8.5      6.5
```

## 9 La librería Matplotlib

### 9.1 La librería Matplotlib

**Matplotlib** es una librería de Python especializada en la creación de gráficos en dos dimensiones.



**Figura 4:** Gráfico con matplotlib

#### 9.1.1 Tipos de gráficos

Permite crear y personalizar los tipos de gráficos más comunes, entre ellos:

- Diagramas de barras
- Histograma
- Diagramas de sectores
- Diagramas de caja y bigotes
- Diagramas de violín
- Diagramas de dispersión o puntos
- Diagramas de líneas
- Diagramas de áreas
- Diagramas de contorno
- Mapas de color

y combinaciones de todos ellos.

En la siguiente [galería de gráficos](#) pueden apreciarse todos los tipos de gráficos que pueden crearse con esta librería.

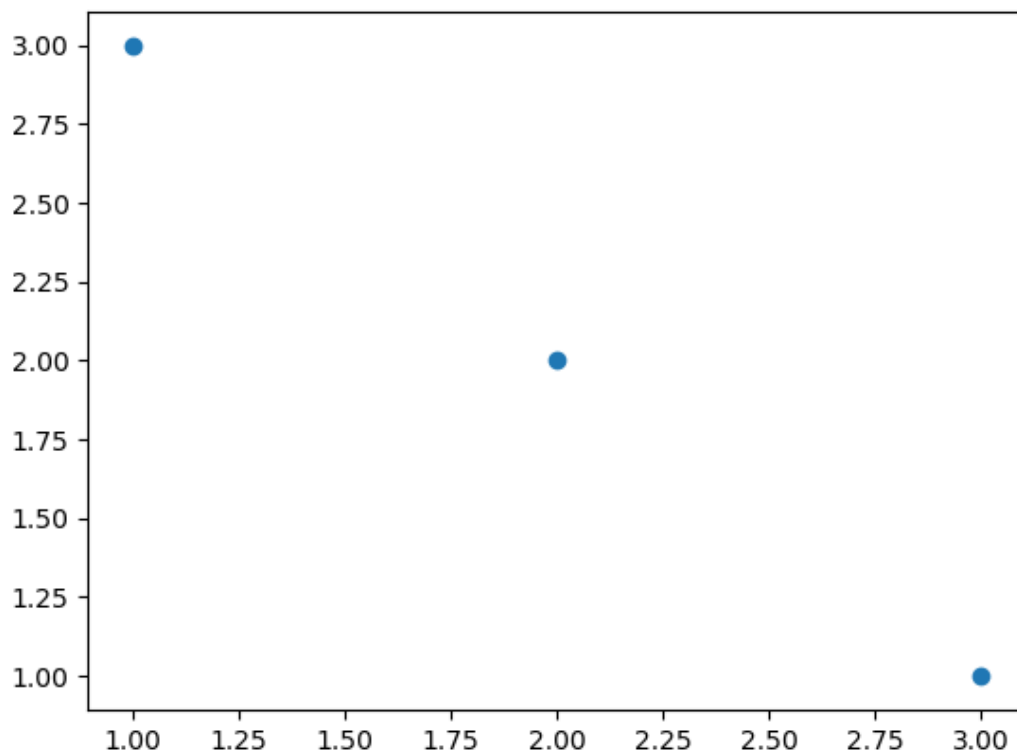
#### 9.1.2 Creación de gráficos con matplotlib

Para crear un gráfico con matplotlib es habitual seguir los siguientes pasos:

1. Importar el módulo `pyplot`.
2. Definir la figura que contendrá el gráfico, que es la region (ventana o página) donde se dibujará. Para ello se utiliza la función `figure()`.
3. Definir los ejes sobre los que se dibujarán los datos. Para ello se utiliza la función `add_axes()` o bien `add_subplot()`.

4. Dibujar los datos sobre los ejes. Para ello se utilizan distintas funciones dependiendo del tipo de gráfico que se quiera.
5. Personalizar el gráfico. Para ello existen multitud de funciones que permiten añadir un título, una leyenda, una rejilla, cambiar colores o personalizar los ejes.
6. Guardar el gráfico. Para ello se utiliza la función `savefig()`.
7. Mostrar el gráfico. Para ello se utiliza la función `show()`.

```
1 # Importar el módulo pyplot con el alias plt
2 import matplotlib.pyplot as plt
3 # Crear la figura
4 fig = plt.figure()
5 # Crear los ejes
6 ax = fig.add_subplot(111)
7 # Dibujar puntos
8 ax.scatter(x = [1, 2, 3], y = [3, 2, 1])
9 # Guardar el gráfico en formato png
10 plt.savefig('diagrama-dispersion.png')
11 # Mostrar el gráfico
12 plt.show()
```

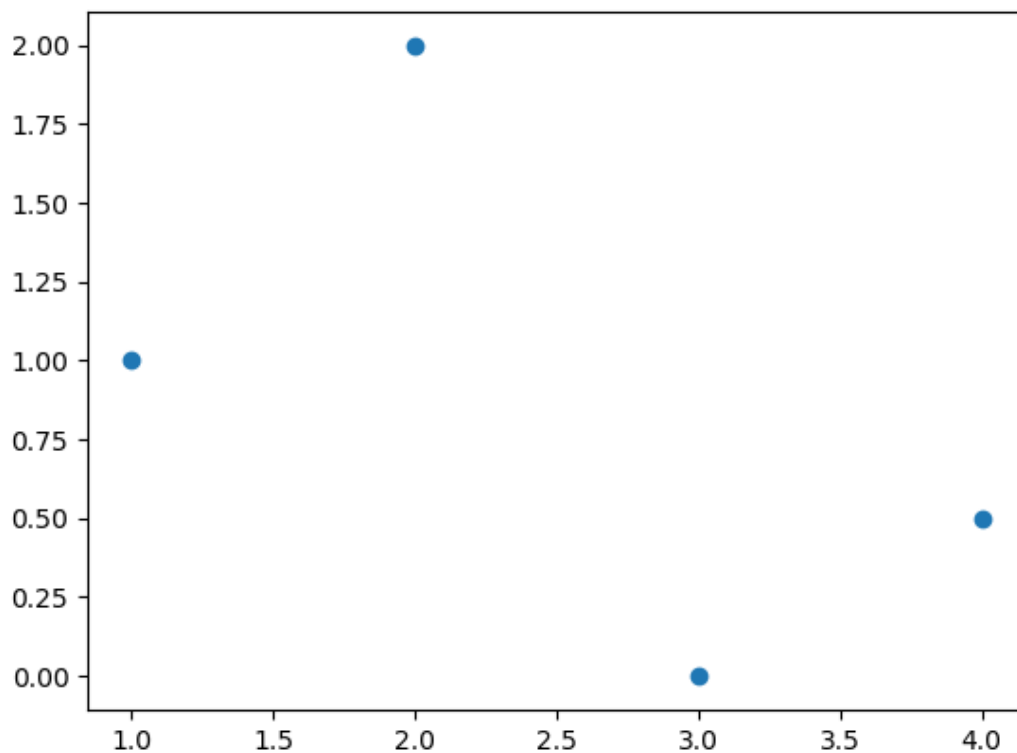


**Figura 5:** Gráfico con matplotlib

### 9.1.3 Diagramas de dispersión o puntos

- `scatter(x, y)`: Dibuja un diagrama de puntos con las coordenadas de la lista `x` en el eje X y las coordenadas de la lista `y` en el eje Y.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 ax.scatter([1, 2, 3, 4], [1, 2, 0, 0.5])
5 plt.show()
```



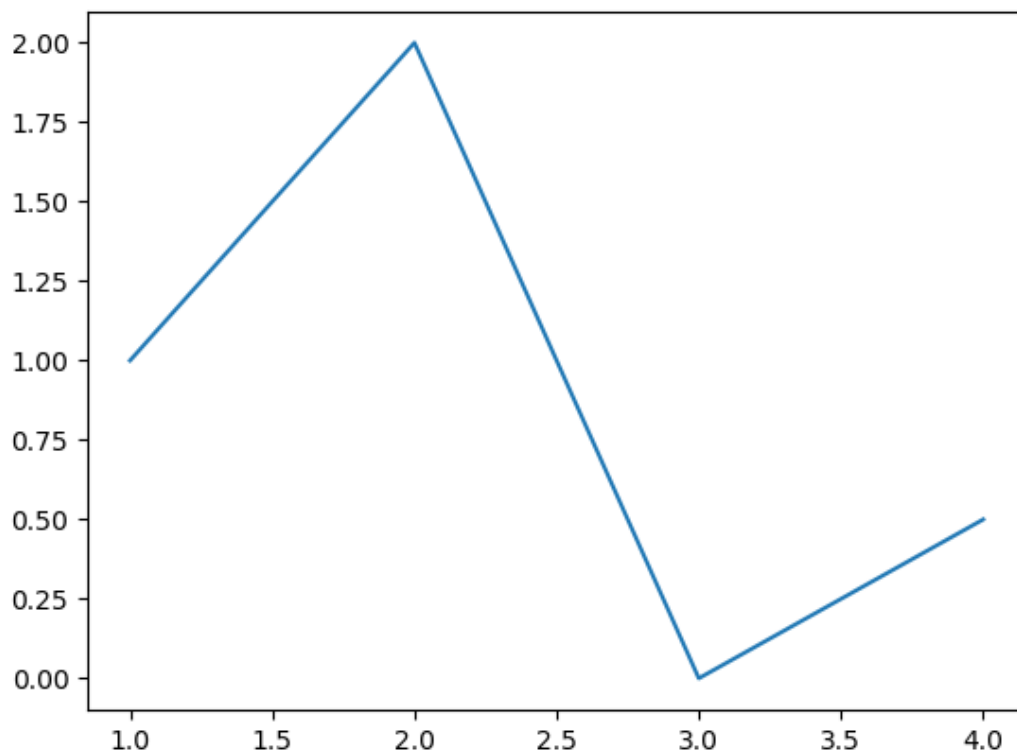
**Figura 6:** Grafico con matplotlib

#### 9.1.4 Diagramas de líneas

- `plot(x, y)`: Dibuja un polígono con los vértices dados por las coordenadas de la lista `x` en el eje X y las coordenadas de la lista `y` en el eje Y.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 ax.plot([1, 2, 3, 4], [1, 2, 0, 0.5])
5 plt.show()
```



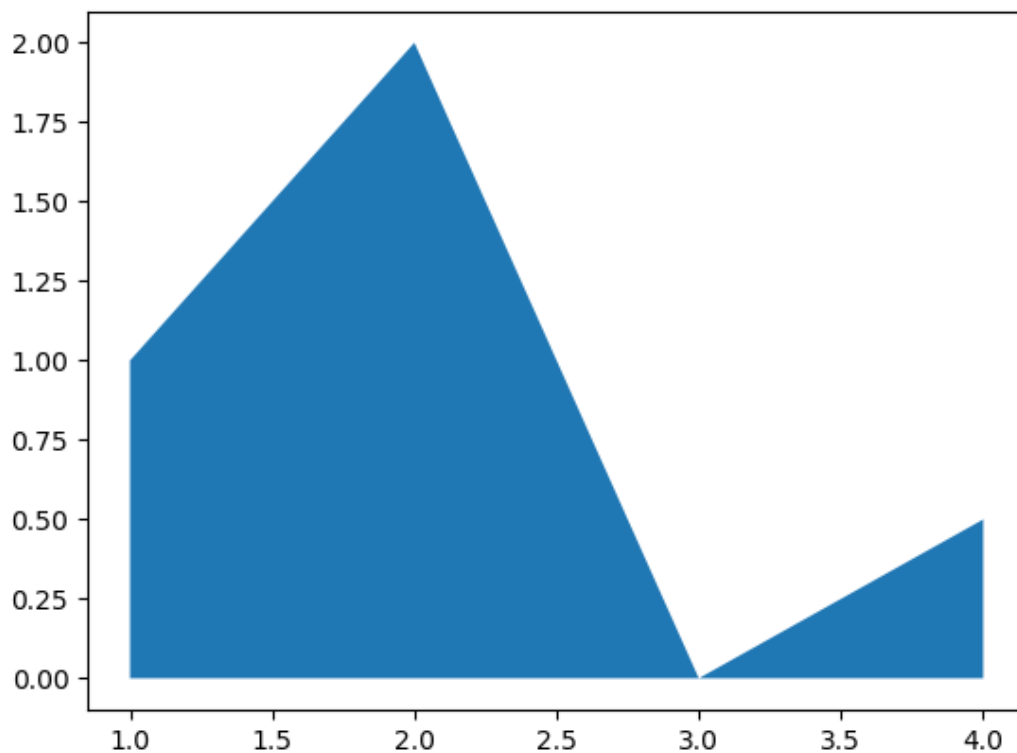


**Figura 7:** Grafico con matplotlib

### 9.1.5 Diagramas de areas

- `fill_between(x, y)`: Dibuja el area bajo el polígono con los vértices dados por las coordenadas de la lista `x` en el eje X y las coordenadas de la lista `y` en el eje Y.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 ax.fill_between([1, 2, 3, 4], [1, 2, 0, 0.5])
5 plt.show()
```

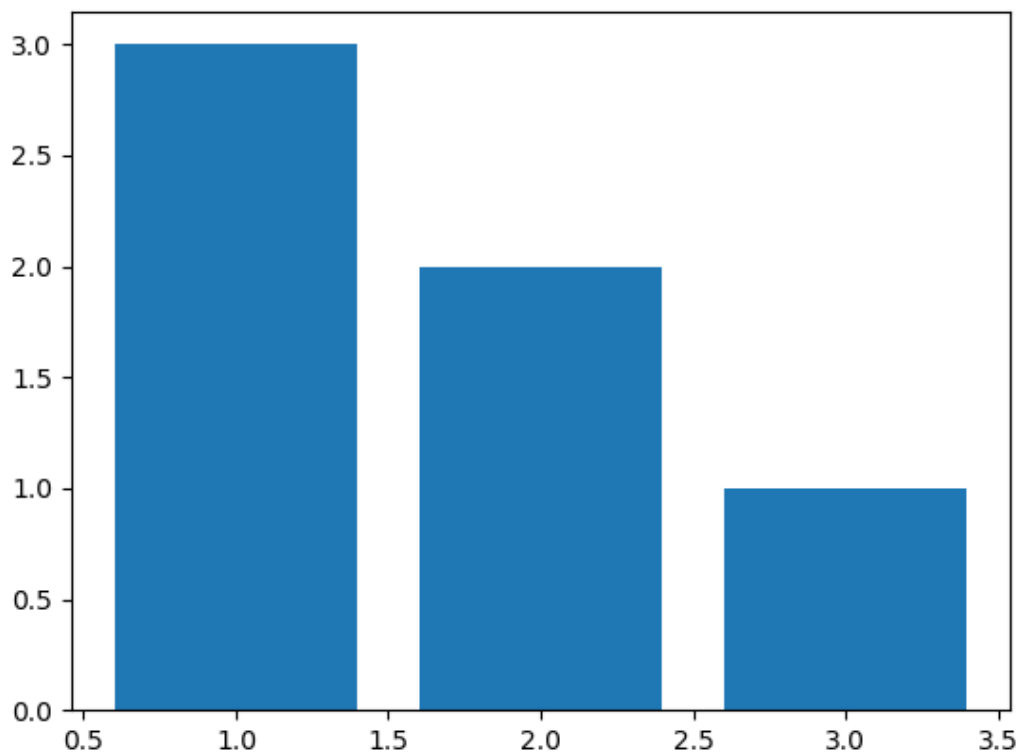


**Figura 8:** Grafico con matplotlib

### 9.1.6 Diagramas de barras verticales

- `bar(x, y)`: Dibuja un diagrama de barras verticales donde `x` es una lista con la posición de las barras en el eje X, e `y` es una lista con la altura de las barras en el eje Y.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 ax.bar([1, 2, 3], [3, 2, 1])
5 plt.show()
```

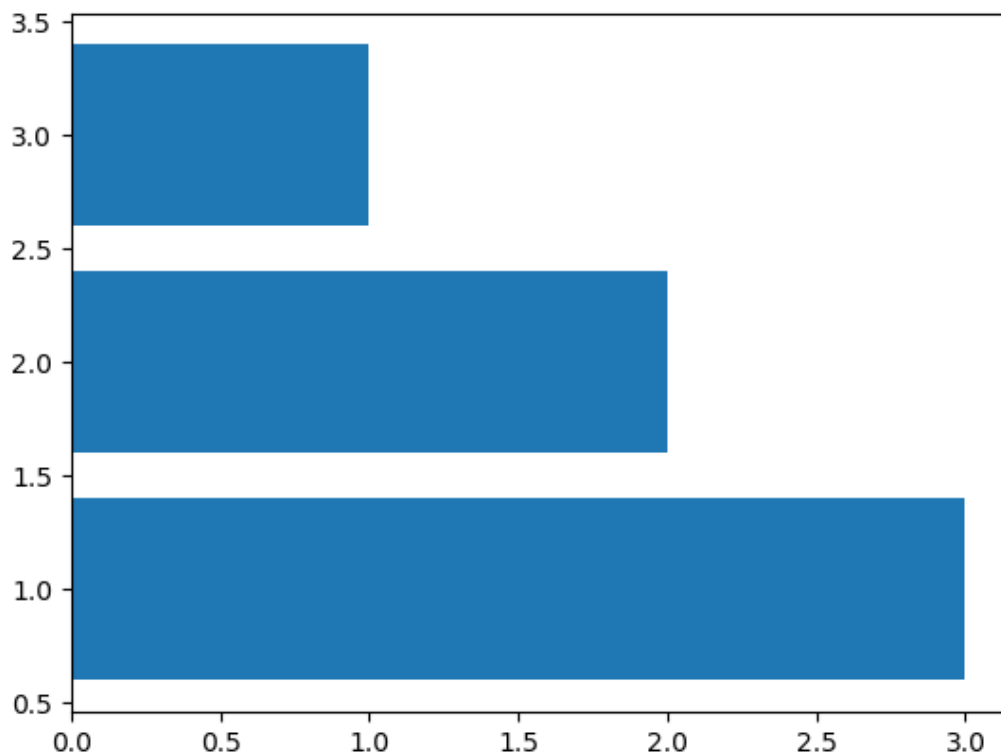


**Figura 9:** Grafico con matplotlib

### 9.1.7 Diagramas de barras horizontales

- `barh(x, y)`: Dibuja un diagrama de barras horizontales donde `x` es una lista con la posición de las barras en el eje Y, y `y` es una lista con la longitud de las barras en el eje X.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 ax.barh([1, 2, 3], [3, 2, 1])
5 plt.show()
```

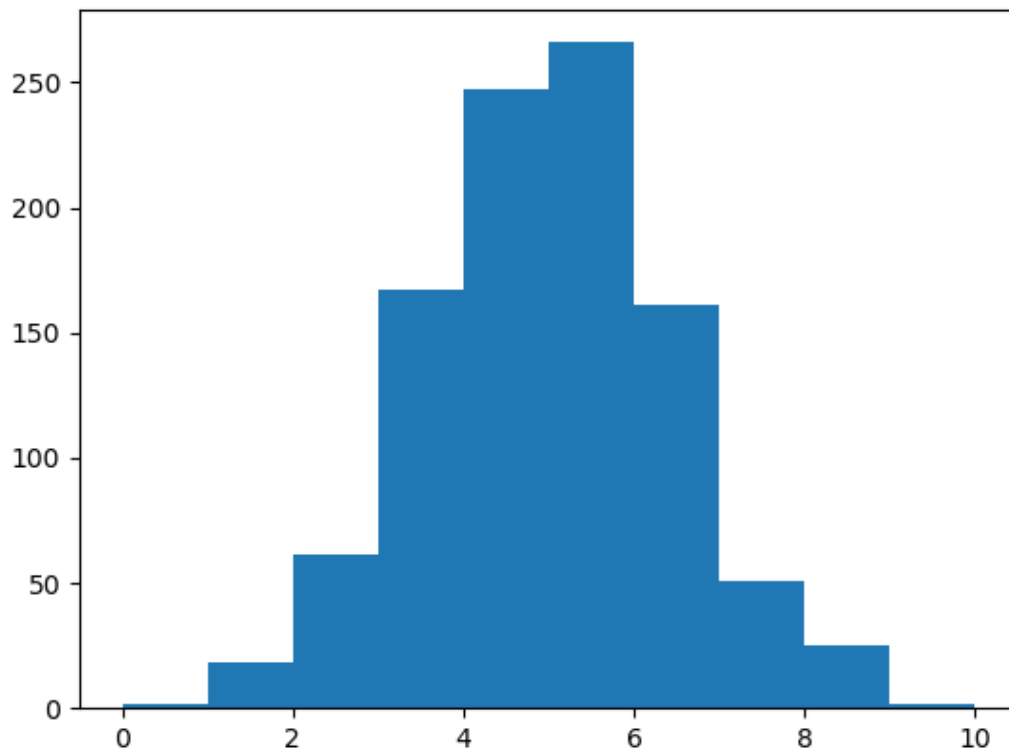


**Figura 10:** Grafico con matplotlib

### 9.1.8 Histogramas

- `hist(x, bins)`: Dibuja un histograma con las frecuencias resultantes de agrupar los datos de la lista `x` en las clases definidas por la lista `bins`.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 fig = plt.figure()
4 ax = fig.add_subplot(111)
5 x = np.random.normal(5, 1.5, size=1000)
6 ax.hist(x, np.arange(0, 11))
7 plt.savefig('histograma.png')
8 plt.show()
```

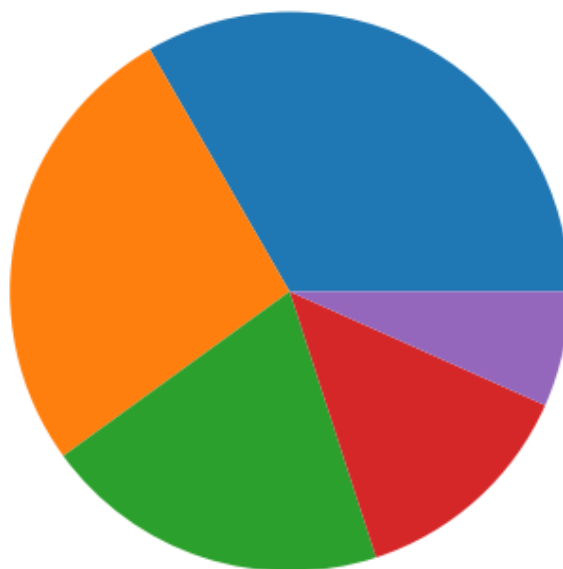


**Figura 11:** Grafico con matplotlib

### 9.1.9 Diagramas de sectores

- `pie(x)`: Dibuja un diagrama de sectores con las frecuencias de la lista `x`.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 ax.pie([5, 4, 3, 2, 1])
5 plt.savefig('diagrama-sectores.png')
6 plt.show()
```

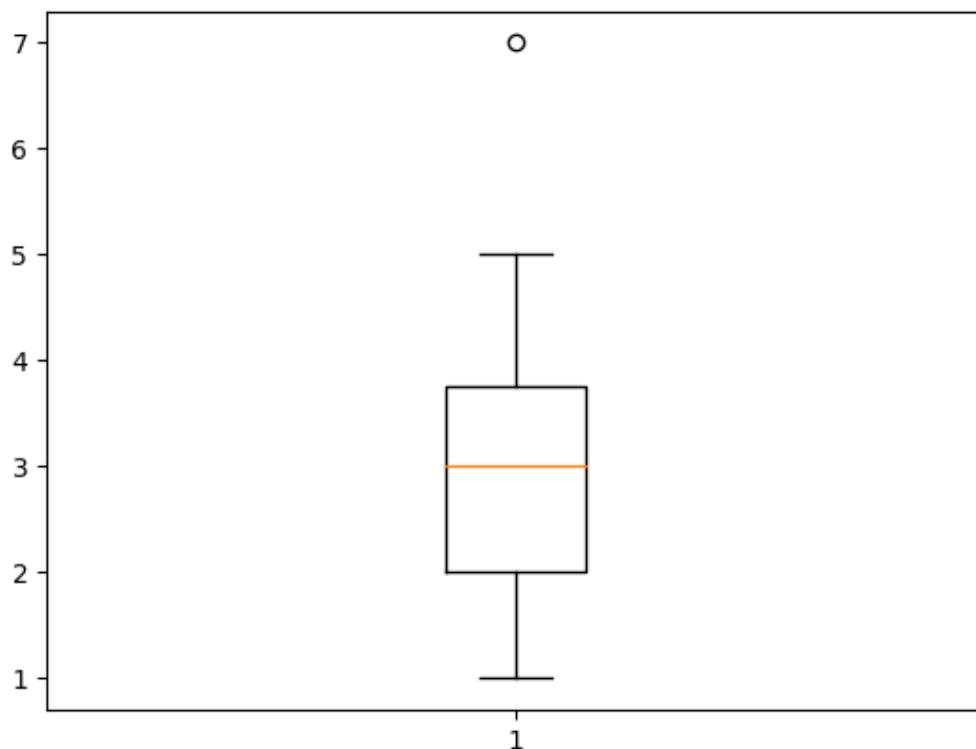


**Figura 12:** Grafico con matplotlib

#### 9.1.10 Diagramas de caja y bigotes

- `boxplot(x)`: Dibuja un diagrama de caja y bigotes con los datos de la lista `x`.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 ax.boxplot([1, 2, 1, 2, 3, 4, 3, 3, 5, 7])
5 plt.savefig('diagrama-sectores.png')
6 plt.show()
```

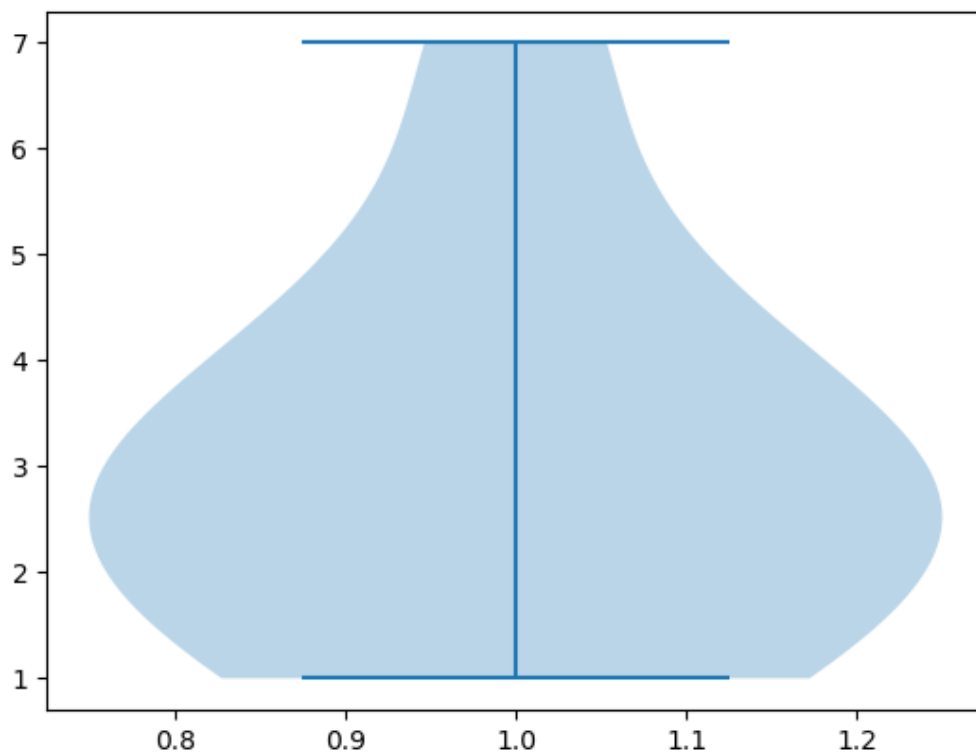


**Figura 13:** Grafico con matplotlib

### 9.1.11 Diagramas de violín

- `violinplot(x)`: Dibuja un diagrama de violín con los datos de la lista `x`.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 ax.violinplot([1, 2, 1, 2, 3, 4, 3, 3, 5, 7])
5 plt.savefig('diagrama-sectores.png')
6 plt.show()
```



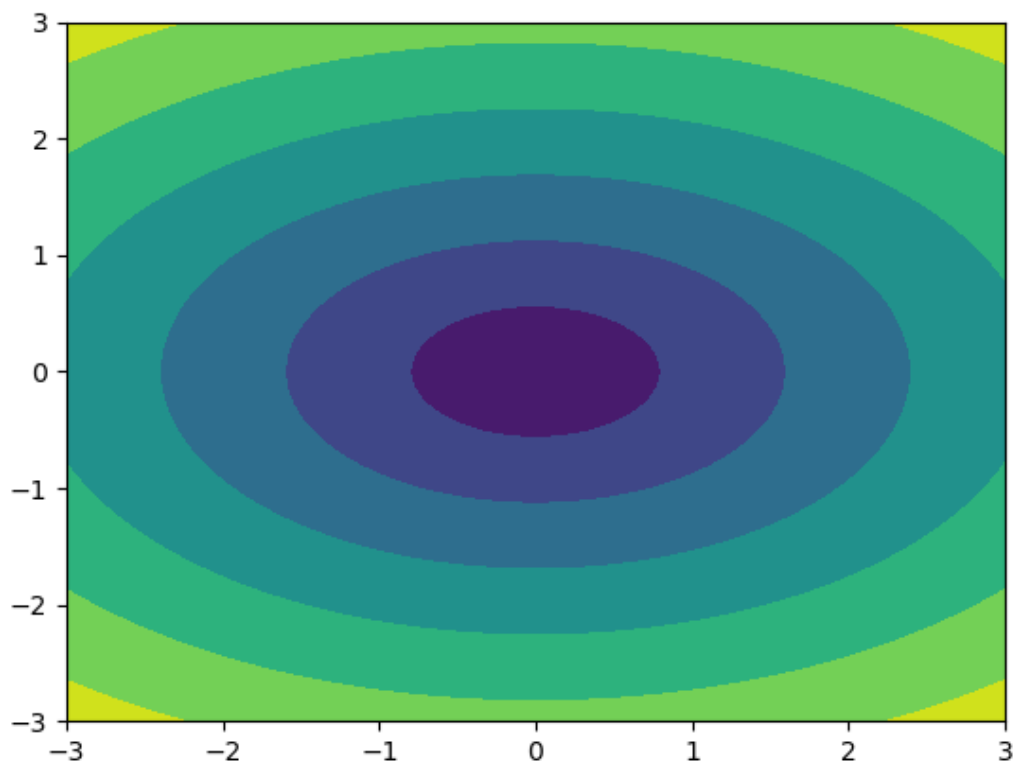
**Figura 14:** Grafico con matplotlib

### 9.1.12 Diagramas de contorno

- `contourf(x, y, z)`: Dibuja un diagrama de contorno con las curvas de nivel de la superficie dada por los puntos con las coordenadas de las listas `x`, `y` y `z` en los ejes X, Y y Z respectivamente.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 x = np.linspace(-3.0, 3.0, 100)
5 y = np.linspace(-3.0, 3.0, 100)
6 x, y = np.meshgrid(x, y)
7 z = np.sqrt(x**2 + 2*y**2)
8 ax.contourf(x, y, z)
9 plt.show()
```



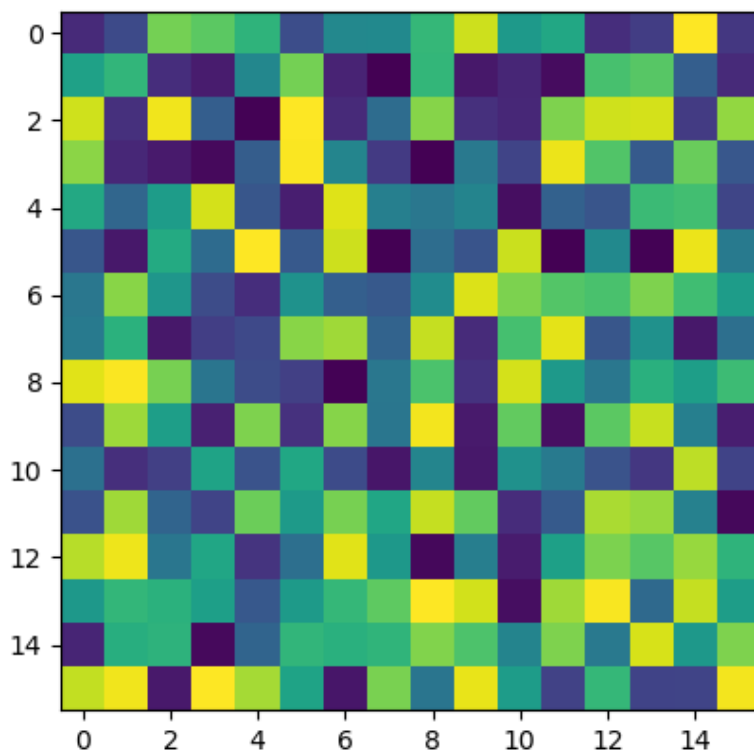


**Figura 15:** Grafico con matplotlib

### 9.1.13 Mapas de color

- `imshow(x)`: Dibuja un mapa de color a partir de una matriz (array bidimensional) `x`.

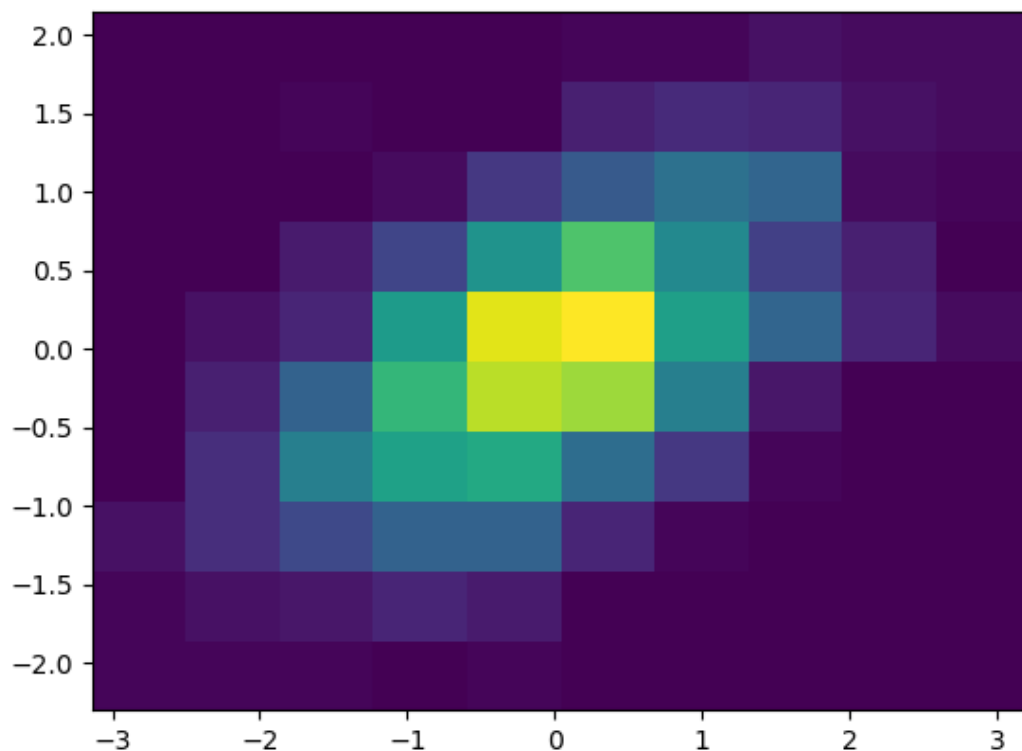
```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 x = np.random.random((16, 16))
5 ax.imshow(x)
6 plt.show()
```



**Figura 16:** Grafico con matplotlib

- `hist2d(x, y)`: Dibuja un mapa de color que simula un histograma bidimensional, donde los colores de los cuadrados dependen de las frecuencias de las clases de la muestra dada por las listas `x` e `y`.

```
1 import matplotlib.pyplot as plt
2 fig = plt.figure()
3 ax = fig.add_subplot(111)
4 x, y = np.random.multivariate_normal(mean=[0.0, 0.0], cov=[[1.0, 0.4],
5                  [0.4, 0.5]], size=1000).T
6 ax.hist2d(x, y)
7 plt.show()
```



**Figura 17:** Grafico con matplotlib

## 10 Apéndice: Depuración de código

### 10.1 Depuración de programas

La depuración es una técnica que permite *trazar* un programa, es decir, seguir el flujo de ejecución de un programa paso a paso, ejecutando una instrucción en cada paso, y observar el estado de sus variables.

Cuando un programa tiene cierta complejidad, la depuración es imprescindible para detectar posibles errores.

Python dispone del módulo `pyd` para depurar programas, pero es mucho más cómodo utilizar algún entorno de desarrollo que incorpore la depuración, como por ejemplo Visual Studio Code.

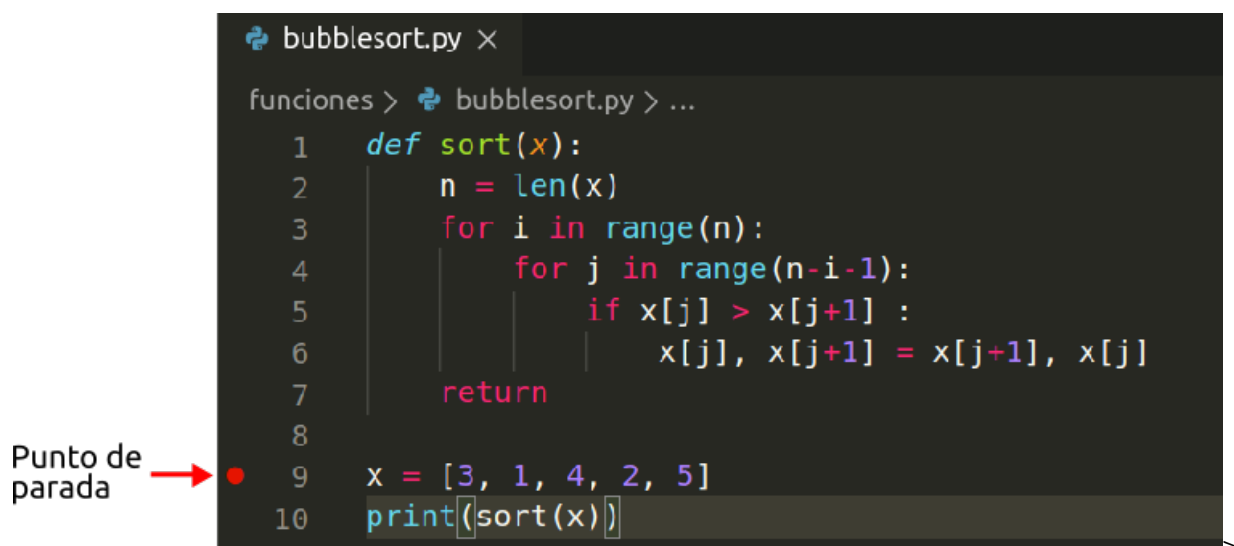
#### 10.1.1 Comandos de depuración


- **Establecer punto de parada:** Detiene la ejecución del programa en una línea concreta de código.

- **Continuar la ejecución:** Continúa la ejecución del programa hasta el siguiente punto de parada o hasta que finalice.
- **Próximo paso:** Ejecuta la siguiente línea de código y para la ejecución.
- **Próximo paso con entrada en función:** Ejecuta la siguiente línea de código. Si se trata de una llamada a una función entonces ejecuta la primera instrucción de la función y para la ejecución.
- **Próximo paso con salida de función:** Ejecuta lo que queda de la función actual y para la ejecución.
- **Terminar la depuración:** Termina la depuración.

### 10.1.2 Depuración en Visual Studio Code

Antes de iniciar la depuración de un programa en VSCode hay que establecer algún punto de parada. Para ello basta con hacer click en le margen izquierdo de la ventana con del código a la altura de la línea donde se quiere parar la ejecución del programa.

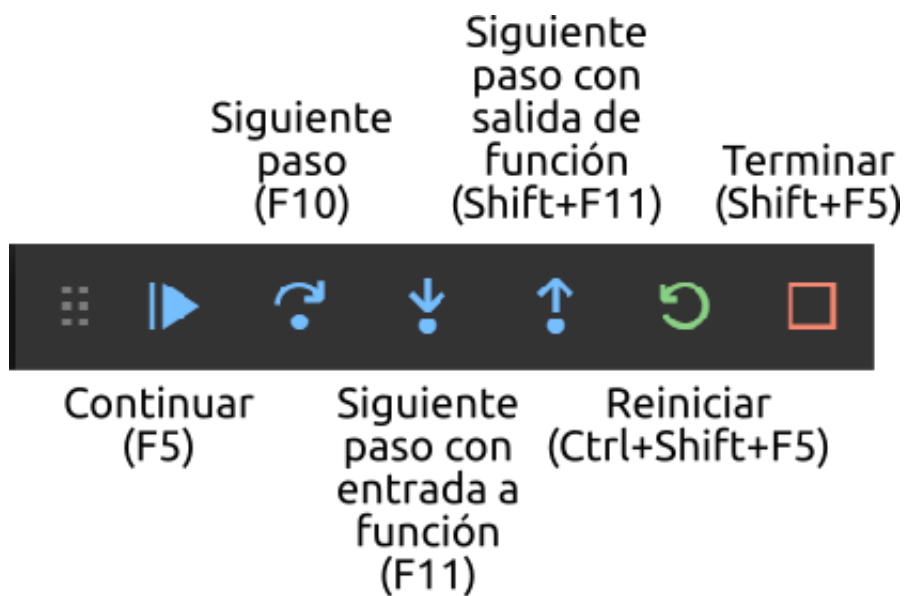


Para iniciar la depuración de un programa en VSCode hay que hacer clic sobre el botón  o pulsar la combinación de teclas (Ctrl+Shift+D).

La primera vez que depuremos un programa tendremos que crear un fichero de configuración del depurador (`launch.json`). Para ello hay que hacer clic en el botón **Run and Debug**. VSCode mostrará los distintos ficheros de configuración disponibles y debe seleccionarse el más adecuado para el tipo de programa a depurar. Para programas simples se debe seleccionar **Python file**.

La depuración comenzará iniciando la ejecución del programa desde el inicio hasta el primer punto de parada que encuentre.

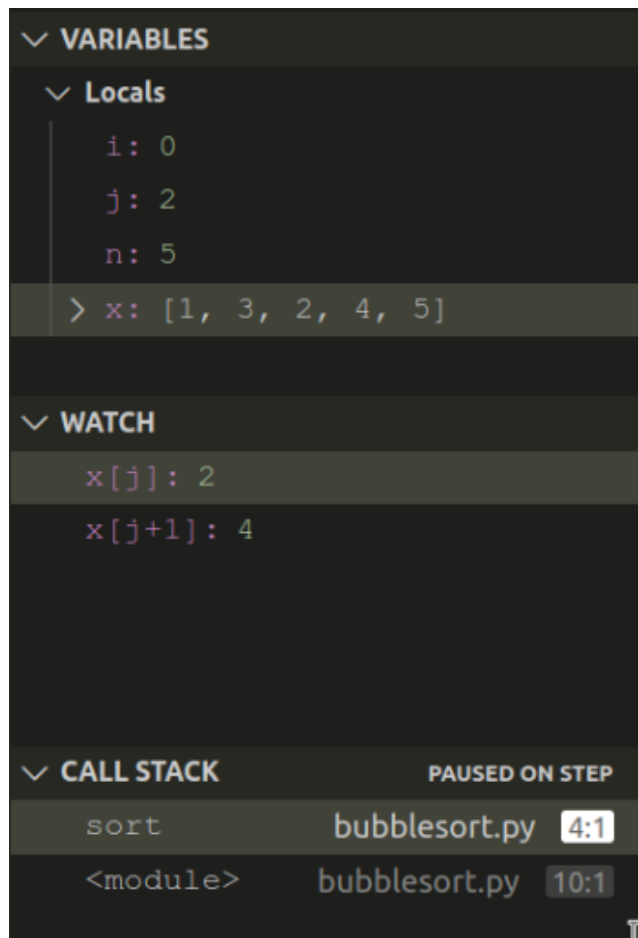
Una vez iniciado el proceso de depuración, se puede avanzar en la ejecución del programa haciendo uso de la barra de depuración que contiene botones con los principales comandos de depuración.



**Figura 18:** Barra de depuración de Visual Studio Code

Durante la ejecución del programa, se puede ver el contenido de las variables del programa en la ventana del estado de las variables.

El usuario también puede introducir expresiones y ver cómo se evalúan durante la ejecución del programa en la ventana de vista de expresiones.



The screenshot displays the debugger interface with three main sections:

- VARIABLES**: Shows the state of local variables. Under the 'Locals' tab, the variables are: `i: 0`, `j: 2`, `n: 5`, and `x: [1, 3, 2, 4, 5]`.
- WATCH**: Shows the state of user expressions. The expressions are: `x[j]: 2` and `x[j+1]: 4`.
- CALL STACK**: Shows the call stack with the status 'PAUSED ON STEP'. The stack contains:
  - `sort` in `bubblesort.py` at line `4:1`.
  - `<module>` in `bubblesort.py` at line `10:1`.

Labels to the right of the screenshot identify each section: 'Estado de las variables locales' for VARIABLES, 'Estado de expresiones del usuario' for WATCH, and 'Pila de llamadas a funciones' for CALL STACK.

# Bibliografía

## 10.2 Referencias

Webs:

- [Python](#) Sitio web de Python.
- [Repl.it](#) Entorno de desarrollo web para varios lenguajes, incluido Python.
- [Python tutor](#) Sitio web que permite visualizar la ejecución del código Python.

Libros y manuales:

- [Tutorial de Python](#) Tutorial rápido de python.
- [Python para todos](#) Libro de introducción a Python con muchos ejemplos. Es de licencia libre.
- [Python para principiantes](#) Libro de introducción Python que abarca orientación a objetos. Es de licencia libre.
- [Python crash course](#) Libro de introducción a Python gratuito.
- [Think python 2e](#). Libro de introducción a Python que abarca también algoritmos, estructuras de datos y gráficos. Es de licencia libre.

- [Learning Python](#) Libro de introducción a Python con enfoque de programación orientada a objetos.

Vídeos:

- [Curso “Python para todos”](#).