

# APRENDE PYTHON DE CERO A EXPERTO

JON VADILLO



Aprende de forma práctica y sencilla uno de los lenguajes más populares de la actualidad.

---

*"Lo único que necesitas saber es que tú puedes aprender cualquier cosa"*  
- Sal Kahn -

# Aprende Python desde cero a experto

Jon Vadillo Romero

Este libro está a la venta en <http://leanpub.com/aprende-python>

Esta versión se publicó en 2020-03-20



Leanpub

Éste es un libro de [Leanpub](#). Leanpub anima a los autores y publicadoras con el proceso de publicación. [Lean Publishing](#) es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

© 2019 - 2020 Jon Vadillo Romero



# **1. Agradecimientos**

*Cada palabra de este libro no hubiese sido posible sin todas aquellas personas que me han acompañado durante este viaje y me han ayudado a tomar decisiones, rumbos y caminos que me han llevado a decidir escribir este libro y ser capaz de hacerlo. Compañeros, amigos, familia: MUCHAS GRACIAS A TODOS.*

*Este libro es una forma de devolver una pequeña parte de todo lo que he recibido y contribuir a que otras personas puedan beneficiarse y acceder de forma libre a una parte del conocimiento que he adquirido a lo largo de los años.*

# Índice general

<b>1. Agradecimientos</b> . . . . .	<b>iv</b>
<b>2. Introducción</b> . . . . .	<b>1</b>
Características principales . . . . .	1
Instalación de Python . . . . .	2
Entorno de desarrollo . . . . .	2
Tu primer programa: ¡Hola Mundo! . . . . .	3
Sintaxis . . . . .	4
Funcionamiento de Python . . . . .	5
<b>3. Variables y Tipos de datos</b> . . . . .	<b>6</b>
Resumen de tipos de variables . . . . .	6
Lectura de datos en Python . . . . .	7
Números . . . . .	7
Cadenas de texto (string) . . . . .	8
Coding time! . . . . .	12
<b>4. Operadores y expresiones</b> . . . . .	<b>14</b>
Operadores aritméticos . . . . .	14
Operadores relacionales o de comparación . . . . .	14
Operadores lógicos . . . . .	15
Coding time! . . . . .	15
<b>5. Estructuras de control</b> . . . . .	<b>17</b>
Condicionales . . . . .	17
Bucles . . . . .	18
Coding time! . . . . .	21
<b>6. Listas y tuplas</b> . . . . .	<b>25</b>
Tuplas . . . . .	26
Coding time! . . . . .	27
<b>7. Diccionarios</b> . . . . .	<b>29</b>
Recorrer un diccionario . . . . .	30
Borrar un elemento . . . . .	30

## ÍNDICE GENERAL

Coding time! . . . . .	31
<b>8. Funciones . . . . .</b>	<b>33</b>
Definir y llamar a una función . . . . .	33
Funciones con argumentos múltiples . . . . .	34
Ámbito de las variables (scope) . . . . .	34
Coding time! . . . . .	35
<b>9. Excepciones . . . . .</b>	<b>36</b>
Excepciones comunes . . . . .	36
Coding time! . . . . .	38
<b>10. Clases y objetos . . . . .</b>	<b>39</b>
Atributos de clase vs Atributos de instancia . . . . .	40
Private y protected . . . . .	41
Coding time! . . . . .	42
<b>11. Herencia . . . . .</b>	<b>45</b>
Herencia múltiple . . . . .	46
Coding time! . . . . .	46
<b>12. Módulos y Paquetes . . . . .</b>	<b>48</b>
Módulos . . . . .	48
Paquetes . . . . .	49
<b>13. Próximos pasos . . . . .</b>	<b>50</b>

## 2. Introducción

Python es sin duda alguna uno de los lenguajes de programación más populares que existen hoy en día. En comparación con otros lenguajes de programación, Python presume de tener una **curva de aprendizaje pequeña y un gran potencial**, ya que hablamos de un lenguaje con el que es posible realizar tareas de todo tipo, como por ejemplo:

- Desarrollo de aplicaciones web
- Inteligencia artificial
- Automatización de tareas
- Análisis avanzado de datos

En este libro aprenderás los fundamentos básicos del lenguaje de forma que al terminar su lectura cuentas con una sólida base sobre la que seguir tu carrera como programador.

### Características principales

Se trata de un lenguaje **open source** en el que destaca su legibilidad. El código es limpio y ordenado, lo cual convierte a Python en un lenguaje cómodo de leer y escribir. En definitiva: **un lenguaje de programación fácil de entender y aprender**.

Al contrario que otros lenguajes de programación como C o Java, **Python es un lenguaje interpretado**, lo que significa que no es necesario compilar el código de Python antes de ejecutarlo. El intérprete irá analizando y ejecutando el código línea por línea.

Otra de las principales características de Python es que es que **es un lenguaje de programación dinámicamente tipado**, es decir, el programador no tiene que declarar el tipo de las variables si no que este se deduce automáticamente en el tiempo de ejecución.

```
1 # La variable "edad" guarda el valor como integer
2 edad = 25
3 print("La variable 'edad' es de tipo: " + str(type(edad)))
4 # Ahora "edad" guarda un string
5 edad = "Tengo 25 años"
6 print("La variable 'edad' es de tipo: " + str(type(edad)))
```

Por último, comentar que una de las mayores ventajas de este **lenguaje de programación orientado a objetos** es la extensa variedad de **liberías y frameworks** disponibles para cualquier propósito, lo cual hace que Python sea la opción perfecta para el desarrollo de aplicaciones de cualquier tipo.

## Instalación de Python

Python es un **lenguaje multiplataforma**, lo que significa que podemos trabajar con Python tanto en Windows, Mac o Linux. A pesar de que todavía encontrarás código escrito en Python 2, en la actualidad la **versión recomendada es Python 3**. A continuación podrás ver cómo instalar Python en cada uno de los entornos.

### Instalar Python en Windows

Sigue los siguientes pasos para instalar Python en Windows:

1. Descarga la última versión de Python para Windows desde la página web oficial: <https://www.python.org/downloads/windows/>
2. En función de la versión de tu sistema operativo (32 bits o 64 bits), tendrás que escoger entre una de las dos versiones: **Windows x86 executable installer** o **Windows x86-64 executable installer**.
3. Una vez descargado, ejecuta el instalador marcando las casillas “Add Python 3.6 to PATH” y “Add Python to your environment variables”.

### Instalar Python en Ubuntu

En primer lugar comprueba que Python no esté instalado en el sistema mediante el siguiente comando:

- ```
1 $ python3 --version
2 Python 3.6.1
```

En caso de no estar instalado, basta con ejecutar el siguiente comando en la consola:

- ```
1 $ sudo apt install python3
```

### Instalar Python en Mac

La instalación para Mac es sencilla y directa. Simplemente descarga la última versión de Python desde la [página web oficial](#)<sup>2</sup> y ejecuta el archivo .pkg descargado.

## Entorno de desarrollo

### Editores de código

Para programar en Python es suficiente con tener un **editor de texto cualquiera**, aunque hoy en día es recomendable utilizar un editor avanzado o un IDE que permita programar de forma ágil.

Estas son algunas de las opciones más recomendadas:

---

<sup>1</sup><https://www.python.org/downloads/windows/>

<sup>2</sup><https://www.python.org/downloads/mac-osx/>

## Editores de texto

- Atom: <https://atom.io/><sup>3</sup>
- Sublime Text: <https://www.sublimetext.com/3><sup>4</sup>
- Visual Studio Code: <https://code.visualstudio.com/><sup>5</sup>

## IDEs

- PyCharm: <https://www.jetbrains.com/pycharm/><sup>6</sup>
- Eclipse (con el plug-in Pydev): <https://www.eclipse.org/><sup>7</sup>; [<https://www.pydev.org/>] (<https://www.pydev.org/>)

## Entorno de desarrollo avanzado

Más adelante, una vez conozcas los fundamentos del lenguaje y tengas fluidez a la hora de programar, te recomendamos profundizar en los conocidos como “entornos Virtuales”. Los entornos virtuales permiten configurar tu entorno de desarrollo de forma más avanzada, creando ambientes (entornos) aislados que permitan trabajar con distintas versiones de frameworks y librerías. Mientras tanto, te recomiendo que continúes paso a paso y en orden por el resto de lecciones.

## Tu primer programa: ¡Hola Mundo!

Ya tenemos Python instalado en nuestro sistema operativo, por lo que estamos preparados para comenzar a escribir y ejecutar código Python. Veremos cómo ejecutar código Python **desde la consola directamente o desde un archivo** de extensión `.py`

## Consola de Python

La consola de Python nos **permite ejecutar código escrito en Python directamente**, sin tener que escribir el código previamente en ningún fichero. Para entrar en la consola de Python, abre una consola (también conocida como terminal o shell) del sistema operativo en el que te encuentres y escribe `py` o `python`. Esto nos mostrará la versión de Python instalada y dará comienzo a la consola de Python:

---

<sup>3</sup><https://atom.io/>

<sup>4</sup><https://www.sublimetext.com/3>

<sup>5</sup><https://code.visualstudio.com/>

<sup>6</sup><https://www.jetbrains.com/pycharm/>

<sup>7</sup><https://www.eclipse.org/>

```
1 python
2 Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel\
3 1)] on win32
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>>
```

Verás cómo el cursor cambia y ahora aparece el símbolo `>>>`. A partir de ahora ya podemos ir ejecutando las instrucciones que queramos. En este caso escribiremos nuestro primer programa, el ya por todos conocido ¡Hola Mundo!:

```
1 >>> print(" ¡Hola Mundo!")
2 ¡Hola Mundo!
```

También puedes ejecutar otras instrucciones. Prueba a ejecutar la siguiente operación:

```
1 >>> 3*3
2 9
```

Para salir de la consola de Python, escribe `exit()` o pulsa CTRL + Z y ENTER.

## Ejecutar fichero

También podemos escribir nuestro código en un **fichero con extensión .py** y ejecutarlo. Abre un editor de texto y crea un fichero llamado `holamundo.py` con el siguiente contenido:

```
1 print(" ¡Hola Mundo!")
```

Para ejecutar el archivo, abre una consola de comandos en la misma ubicación donde has guardado el archivo y escribe `python` seguido del nombre del fichero:

```
python holamundo.py
```

¡Enhorabuena! Ya has escrito y ejecutado tu primer programa en Python.

## Sintaxis

### Indentación

Un aspecto muy importante a mencionar antes de comenzar a programar en Python es el hecho de que **Python utiliza la indentación** (también conocida como sangría, tabulación o espaciado) para **delimitar los bloques de código**. La indentación estándar de Python requiere una tabulación de 4 espacios:

```
1 x = 5
2 if x == 5:
3     # tabulación de 4 espacios
4     print("El valor de x es 5.")
```

En caso contrario, al ejecutar nuestro código recibiremos un error como el siguiente

```
1 IndentationError: unindent does not match any outer indentation level
```

## Comentarios

A la hora de programar es posible que queramos introducir en el código comentarios que añadan información extra sin afectar a la ejecución del programa. En Python los comentarios se insertan mediante el carácter hash (#):

```
1 # Mi primer comentario
2 x = 5 # Un comentario junto con el código
```

## Funcionamiento de Python

Python ejecuta nuestro código línea por línea. Por cada línea de código estas son las acciones que se realizan:

1. **Analizar (*parse* en inglés) el código** comprobando que formato y la sintaxis es correcta, es decir, que cumplen las normas establecidas para el lenguaje de programación.
2. **Traducir el código a bytecode** (instrucciones que nuestra máquina puede ejecutar).
3. **Enviar el código para su ejecución** a la Python Virtual Machine(PVM), donde el código es ejecutado.

# 3. Variables y Tipos de datos

Las variables permiten **almacenar datos del programa**. Estas serán de un tipo u otro en función de la información que se guarde en ellas.

```
1 nombre = 'Amaia' # cadena de texto
2 edad = 30 # número entero
```

El nombre de una variable se conoce como **identificador**, y deberá cumplir las siguientes reglas:

- Comenzar con una letra o un guión bajo.
- El resto del nombre estará formado por letras, números o guiones bajos.
- Los nombres de las variables son *case sensitive*, es decir, no es lo mismo que una variable se llame resultado que RESULTADO.
- Existen una serie de palabras reservadas que no se pueden utilizar (def, global, return, if, for, ...).

Algunas de las recomendaciones respecto a los nombres de las variables están recogidas en la [Guía oficial de Estilos PEP8 de Python](#)<sup>1</sup>. Entre las más habituales encontramos las siguientes:

- Utilizar nombres descriptivos, en minúsculas y separados por guiones bajos si fuese necesario: resultado, mi\_variable, valor\_anterior,...
- Escribir las constantes en mayúsculas: MI\_CONSTANTE, NUMERO\_PI, ...
- Antes y después del signo =, debe haber uno (y solo un) espacio en blanco.

**Nota:** No olvides que lo que la guía plantea son recomendaciones y no obligaciones. Por ejemplo, mientras PEP8 recomienda tabular el código con 4 espacios en blanco, la guía particular de los desarrolladores de Google habla de 2 espacios en lugar de 4.

## Resumen de tipos de variables

---

<sup>1</sup><https://www.python.org/dev/peps/pep-0008/>

```
1     edad = 24 # número entero (integer)
2     precio = 112.9 # número de punto flotante (float)
3     titulo = 'Aprende Python desde cero' # cadena de texto (string)
4     test = True # booleano
```

## Lectura de datos en Python

La función `input()` permite introducir datos al usuario:

```
1 >>> nombre = input()
2 Leire
3 >>> print(nombre)
4 Leire
```

Como se puede ver en el siguiente ejemplo, es posible también mostrar un mensaje al usuario, tal y como muestra el siguiente ejemplo.

```
1 >>> nombre = input("Escribe tu nombre: ")
2 Escribe tu nombre: Leire
3 >>> print(nombre)
4 Leire
```

## Números

Python soporta dos tipos de números: enteros (`integer`) y de punto flotante (`float`).

```
1 # integer
2 x = 5
3 print(x)
4
5 # float
6 y = 5.0
7 print(y)
8
9 # Otra forma de declarar un float
10 z = float(5)
11 print(z)
```

Si tenemos dudas del valor de una variable, podemos mostrar su tipo utilizando la función `type()`:

```
1 >>> x = 5.5
2 >>> type(x)
3 <class 'float'>
```

## Cadenas de texto (string)

Las cadenas de texto o strings se definen mediante comilla simple (' ') o doble comilla (" "):

```
1 mi_nombre = 'Ane'
2 print(mi_nombre)
3 mi_nombre= "Ane"
4 print(mi_nombre)
```

La diferencia principal se encuentra en que las comillas dobles aportan mayor facilidad en textos que incluyan apóstrofes:

```
1 mi_nombre = 'I\'m John'
2 print(mi_nombre)
3 mi_nombre= "I'm John"
4 print(mi_nombre)
```

Más información sobre strings y caracteres especiales en: <https://docs.python.org/3/tutorial/introduction.html#string>

Para definir strings multi-línea se utiliza la triples comillas (""):

```
1 frase = """ esto es una
2         frase muy larga de más de
3         una línea ... """
```

## Concatenación de strings

Es posible unir dos strings con el operador +:

---

<sup>2</sup><https://docs.python.org/3/tutorial/introduction.html#strings>

```
1 >>> primera_palabra = 'Hola'
2 >>> frase_completa = primera_palabra + ', mundo'
3 >>> print(frase_completa)
4 'Hola, mundo'
5
6 >>> segunda_palabra = 'mundo'
7 >>> frase_completa = primera_palabra + ', ' + segunda_palabra
8 >>> print(frase_completa)
9 'Hola, mundo'
```

### Método alternativo 1: str.join():

El método `join()` recibe como argumento el listado (de tipo List, Tuple, String, Dictionary y Set) de strings que se desean concatenar. Se invoca sobre el separador que se utilizará para unir las cadenas (el cual a su vez es un string también):

```
1 >>> strings = ['do', 're', 'mi']
2 >>> separador = ' - '
3 >>> separador.join(strings)
4 'do - re - mi'
```

Para iterar un elemento detrás del otro se introducirá string vacío como separador:

```
1 >>> strings = ['do', 're', 'mi']
2 >>> ''.join(strings)
3 'doremi'
```

### Método alternativo 2: str.format():

Python 3 introdujo una nueva forma para formatear strings, la cual sustituye a la anterior en la que se hace uso del operador `%`. Para ello se invoca el método `format()` de un string:

```
1 # Ordenado por defecto:
2 frase = "Meses: {}, {} y {}".format('Enero', 'Febrero', 'Marzo')
3 print(frase)
4
5 # Especificar el orden indicando la posición:
6 frase = "Meses: {1}, {0} y {2}".format('Enero', 'Febrero', 'Marzo')
7 print(frase)
8
9 # Especificar el orden mediante parejas clave-valor:
10 frase = "Meses: {ene}, {feb} y {mar}".format(ene='Enero', feb='Febrero', m='Marzo')
11 print(frase)
```

## Cadenas 'f' (f-strings)

La versión 3.6 de Python trajo un gran avance a la hora de integrar variables o expresiones en cadenas de caracteres. Se introdujeron las llamadas *f-strings*, una forma más cómoda y directa para insertar variables y expresiones en cadenas. Permiten introducir cualquier variable o expresión dentro de un string incluyendo la variable entre llaves { y }.

Veamos un ejemplo:

```
1 nombre = "Nora"
2 edad = 22
3 saludo = f"Me llamo {nombre} y tengo {edad} años."
```

Para indicar que se trata de un *f-string*, este deberá incluir la letra 'f' antes del comienzo de la cadena (antes de las comillas). A continuación se muestra otro ejemplo en el que se introduce una expresión:

```
1 a = 4
2 b = 3
3 print(f"La multiplicación de {a} y {b} es igual a {a * b}")
```

## Conversión de tipos

A la hora de concatenar un string con otras variables como integer o float puede haber problemas:

```
1 >>> edad = 25
2 >>> nota_media = 7.3
3 >>> print("Tengo " + edad + " años y mi nota media es " + nota_media + ".")
4
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in <module>
7   TypeError: can only concatenate str (not "int") to str
```

Mediante la función `str()` podemos convertir un valor a string y evitar así cualquier tipo de problema:

```
1 >>> edad = 25
2 >>> nota_media = 7.3
3 >>> print("Tengo " + str(edad) + " años y mi nota media es " + str(nota_media) + ".")
4 Tengo 25 años y mi nota media es 7.3.
```

De igual manera es posible convertir a otros tipos con las funciones `int()`, `float()` and `bool()`.

## Métodos en cadenas de texto (string)

Es posible **obtener un carácter concreto** de un string utilizando los corchetes [] y el índice del carácter al que queremos acceder:

```
1 frase = 'Aprendiendo a programar en Python'
2 frase[0] # devuelve el primer caracter
3 frase[1] # devuelve el segundo caracter
4 frase[-1] # devuelve el primer caracter empezando por el final
5 frase[-2] # # devuelve el segundo caracter empezando por el final
```

Si queremos **obtener un substring**, utilizaremos la siguiente notación:

```
1 frase = 'Aprendiendo a programar en Python'
2 mi_substring = frase[1:5]
3 # devolverá los caracteres desde la posición 1 hasta la 5 (no incluye el 5)
```

En caso de dejar la primera variable vacía, se considera la primera posición del string. Dejando la segunda variable vacía se considera la última posición del string:

```
1 >>> frase = 'Aprendiendo a programar en Python'
2 >>> mi_substring = frase[:5]
3 >>> mi_substring
4 'Apren'
5 >>> mi_substring = frase[4:]
6 >>> mi_substring
7 'ndiendo a programar en Python'
```

Otros métodos útiles de string:

```
1 len(str) # devuelve la longitud del string
2 str.upper() # convierte a mayúsculas
3 str.lower() # convierte a minúsculas
4 str.title() # convierte a mayúsculas la primera letra de cada palabra
5 str.count(substring [, inicio, fin]) # devuelve el número de veces que aparece
6 # el substring en el string. Opcionalmente se puede indicar el inicio y fin.
7 str.find('d') # devuelve el índice de la primera aparición de 'd'
8 # (devolverá -1 si no lo encuentra)
9 substr in str # devuelve True si el string contiene el substring
10 str.replace(old, new [, count]) # reemplaza 'old' por 'new' un máximo de 'count' vec\
11 es.
12 str.isnumeric() # devuelve True si str contiene solamente números
```

# Coding time!

## Ejercicio 1

Escribe un programa que contenga las siguientes variables:

- nombre: tipo string y valor "Michael Jordan"
- edad: tipo integer y valor 50
- media\_puntos: tipo float y valor 28.5
- activo: False

El programa deberá mostrar en pantalla todos los valores.

## Ejercicio 2

Escribe un programa que solicite el nombre, DNI y edad, lo almacene en 3 variables distintas y muestre por pantalla los valores introducidos.

## Ejercicio 3

Escribe un programa que genere un string compuesto por los primeros 3 caracteres y los últimos 3 caracteres de un string introducido por el usuario. Pista: tendrás que utilizar la función `len()` en la obtención de los últimos 3 caracteres.

- Ejemplo 1: 'aprendiendo'
- Resultado 1: 'aprndo'
- Ejemplo 2: 'escribiendo código'
- Resultado 2: 'escigo'

## Ejercicio 4

Escribe un programa que solicite al usuario dos números y una frase. El primer número introducido se corresponderá a la posición de inicio del substring que deberá mostrar el programa por pantalla. El segundo número indicará la longitud de dicho substring.

- Ejemplo 1: Posicion=4, Longitud=8, Frase='Desarrollar es mi nueva afición'
- Resultado 1: "rrollar "
- Ejemplo 2: Posicion=8, Longitud=11, Frase='Bienvenido a la clase de programación'
- Resultado 2: "do a la cla"

## Ejercicio 5

Escribe un programa que solicite al usuario una frase. A continuación le solicitará la letra que quiere reemplazar y por qué letra deberá reemplazarse. Por último el programa mostrará el número de veces que la letra está presente en la frase y el resultado final tras reemplazarla.

- Ejemplo: 'Desarrollar es mi nuevo pasatiempos', 'a','e'
- Resultado: 4 apariciones. 'Deserroller es mi nueve pesetiempos'

# 4. Operadores y expresiones

Los operadores son símbolos especiales que permiten realizar operaciones aritméticas o lógicas.

## Operadores aritméticos

Los operadores aritméticos se utilizan para realizar operaciones matemáticas (suma, resta, multiplicación,...). La tabla siguiente contiene todos los operadores aritméticos permitidos por Python:

Operador	Ejemplo	Significado
+	a + b	Suma
-	a - b	Resta
-	-a	Negación (asignar valor negativo)
*	a * b	Multiplicación
/	a / b	División
%	a % b	Módulo (resto de la división)
//	a // b	División entera
**	a ** b	Exponente

Ejemplos:

```
1 x = 5
2 y = 2
3 print(x + y) # 7
4 print(x - y) # 3
5 print(x * y) # 10
6 print(x / y) # 2.5
```

## Operadores relacionales o de comparación

Los operadores relacionales se utilizan para comparar valores y devuelven como resultado un booleano: True o False.

Operador	Ejemplo	Significado
>	a > b	Mayor que: True si a es mayor que b
<	a > b	Menor que: True si a es menor que b
==	a == b	Igual: True si a y b son iguales
!=	a != b	Distinto: True si a y b son distintos
>=	a >= b	Mayor o igual: True si a es igual o mayor que b
<=	a >= b	Menor o igual: True si a es igual o menor que b

## Operadores lógicos

Los operadores lógicos and or, y not evalúan valores devolviendo también True o False como resultado:

Operador	Ejemplo	Significado
and	a and b	True si a y b son True
or	a or b	True si a o b son true
not	not b	True si b es falso

## Coding time!

### Ejercicio 1

Crea un programa que solicite al usuario un número del 1 al 10 y muestre por pantalla la tabla de multiplicación del 1 al 10.

Ejemplo:

```

1 Introduce un número del 1 al 10: 3
2 3 x 1 = 3
3 3 x 2 = 6
4 3 x 3 = 9
5 3 x 4 = 12
6 3 x 5 = 15
7 3 x 6 = 18
8 3 x 7 = 21
9 3 x 8 = 24
10 3 x 9 = 27
11 3 x 10 = 30

```

### Ejercicio 2

Crea un programa que solicite al usuario dos números enteros y muestre por pantalla el resultado de las siguientes operaciones: suma, resta, multiplicación y división.

### Ejemplo:

- 1 Introduce el primer número: 8
- 2 Introduce el segundo número: 2
- 3 La suma es: 10
- 4 La resta es: 6
- 5 La multiplicación es: 16
- 6 La división es: 4.0

## Ejercicio 3

Crea un programa que solicite al usuario el radio de un círculo y calcule el área.

Nota: Utiliza 3.14159 como número PI para el cálculo del área.

### Ejemplo:

- 1 Introduce el radio: 3
- 2 El área es: 28.274309999999996

# 5. Estructuras de control

## Condicionales

Las estructuras de control se utilizan para **ejecutar bloques de código en función de condiciones**.

### Sentencia IF - ELSE

Se evalúa la condición especificada en la sentencia `if` y en caso de cumplirse se ejecutará el bloque de código indentado (tabulado). En caso de que el resultado de la condición sea `False`, el bloque especificado no se ejecutará:

```
1 numero = 5
2 if numero > 1:
3     # Se ejecutará cuando la condición sea True
4     print("Es mayor que uno")
```

Las condiciones pueden tener mayor complejidad:

```
1 edad = 16
2 altura = 175
3 if (edad > 14 and altura > 160):
4     print("Puede montarse en la montaña rusa")
```

Mediante la palabra reservada `else` es posible especificar un bloque de código que se ejecute en caso de que la condición no se cumpla:

```
1 numero = 2
2 if numero > 10:
3     # Se ejecutará cuando la condición sea True
4     print("Es mayor que diez")
5 else:
6     # Se ejecutará cuando la condición sera False
7     print("Es menor o igual que diez")
```

También podemos comprobar más condiciones mediante la expresión `elif`. En este caso, se seguirán comprobando todas las condiciones `elif` hasta que una de ellas se cumpla. En caso contrario, se ejecutará el bloque de código dentro de `else` (si lo hubiera).

```
1 numero = 5
2 if numero < 3:
3     print("Es menor que 3")
4 elif numero < 6:
5     print("El número está entre el 3 y el 5")
6 else:
7     print("Es mayor o igual a 6")
```

Tal y como muestra el siguiente código de ejemplo, Python no tiene limitaciones para anidar distintos bloques de IFs.

```
1 numero = 2
2 if numero >= 0:
3     if numero == 0:
4         print("El valor es 0")
5     else:
6         print("Es un número positivo")
7 else:
8     print("Es un número negativo")
```

## Bucles

Los bucles permiten ejecutar un bloque de código tantas veces como queramos.

### Sentencia WHILE

La sentencia `while` permite ejecutar un bloque de código mientras la expresión que definamos se cumpla (es decir, devuelva `True`). Python interpretará como `True` cualquier valor distinto a `0` o `None`.

```
1 contador = 0
2 while(contador < 5):
3     # Se ejecutará mientras la variable contador sea menor a 5.
4     contador = contador+1
5     print("Iteración número",contador)
6 print (" ¡Fin!")
```

Para detener una ejecución de forma voluntaria se utiliza la sentencia `break`:

```
1 contador = 0
2 while(contador < 5):
3     contador = contador+1
4     print("Iteración número",contador)
5     if contador == 3:
6         break
7 print (" ¡Fin!")
```

También es posible saltar únicamente la iteración actual mediante la sentencia `continue`:

```
1 contador = 0
2 while(contador < 5):
3     contador = contador+1
4     if contador == 3:
5         continue
6     print("Iteración número {}".format(contador))
7 print (" ¡Fin!")
```

La salida del programa anterior será la siguiente:

```
1     Iteración número 1
2     Iteración número 2
3     Iteración número 4
4     Iteración número 5
5     Bucle while finalizado
```

Otros lenguajes de programación ofrecen otra estructura similar conocida como `DO - WHILE`. No es el caso de Python, por lo que habría que emular dicho comportamiento mediante nuestro código.

## Bucle WHILE con ELSE

La expresión `else` puede utilizarse también tras un bloque `while`. De esta forma podemos definir un bloque de código que se ejecutará una vez finalizado el bloque `while` (es decir, cuando la condición se evalúe `False` y no se haya finalizado mediante un `break`):

```
1 count = 0
2 while(count < 5):
3     count = count+1
4     print("Iteración número {}".format(count))
5 else:
6     print("Bucle while finalizado")
```

## Sentencia FOR

A diferencia de otros lenguajes de programación, en Python la sentencia FOR itera únicamente por secuencias (listas, tuplas, cadenas de caracteres,...).

```
1 alumnos = ["Ane", "Mikel", "Unai", "Lorea"]
2 for alumno in alumnos:
3     print(alumno)
```

También es posible utilizarlo para recorrer un string:

```
1 frase = "Aprendiendo Python"
2 for c in frase:
3     print(c)
```

Para detener una ejecución se utiliza la sentencia break:

```
1 numeros = [4,8,2,7,1,9,3,5]
2 total = 0
3
4 for n in numeros:
5     total += n
6     if total > 10:
7         break
```

Al igual que en otras estructuras de repetición, también es posible saltar únicamente la iteración actual mediante la sentencia continue:

```
1 numeros = [4,8,2,7,1,9,3,5]
2 total = 0
3
4 # solo sumar los números impares
5 for num in numeros:
6     if num % 2 == 0:
7         print("Numero par, no lo sumamos")
8         continue
9     total += n
```

## Bucle FOR con ELSE

Python permite definir un bloque de código que se ejecutará una vez finalice la iteración por todos los elementos de una lista. Es importante mencionar que no se ejecutará si se ha finalizado mediante break.

```
1 alumnos = ["Ane", "Mikel", "Unai", "Lorea"]
2 for alumno in alumnos:
3     print(alumno)
4 else:
5     print("No quedan más alumnos.")
```

## La función range()

La función `range([start,] stop [, step])` devuelve una secuencia de números. Es por ello que se utiliza de forma frecuente para iterar:

```
1 for i in range(3):
2     print(i)
3 # 0
4 # 1
5 # 2
```

También podemos indicar el inicio, fin y step:

```
1 print("Números del 5 al 10")
2 for i in range(5, 10):
3     print(i, end=', ')
4 # 5, 6, 7, 8, 9,
5
6 print("Números impares del 1 al 10")
7 for i in range(1, 10, 2):
8     print(i, end=', ')
9 # 1, 3, 5, 7, 9,
```

Para iterar por una lista utilizando el índice, basta con combinarlo con la función `len()`:

```
1 alumnos = ["Ane", "Mikel", "Unai", "Lorea"]
2 for i in range(len(alumnos)):
3     print(alumnos[i])
```

## Coding time!

### Ejercicio 1

Crea un programa que solicite un número al usuario y devuelva el siguiente mensaje:

- Si es mayor que 0: “Es un número positivo.”
- Si es igual a 0: “Es igual a cero.”
- Si es menor que 0: “Es un número negativo.”

Ejemplo 1:

- 1 Introduce un número: 5
- 2 Es un número positivo

Ejemplo 2:

- 1 Introduce un número: -3
- 2 Es un número negativo

## Ejercicio 2

Escribe un programa que solicite dos números enteros al usuario y muestre por pantalla la suma de todos los números enteros que hay entre los dos números (ambos números incluidos).

Ejemplo 1:

- 1 Introduce el número de inicio: 4
- 2 Introduce el número de fin: 8
- 3 El resultado es: 30

Ejemplo 2:

- 1 Introduce el número de inicio: 10
- 2 Introduce el número de fin: 15
- 3 El resultado es: 75

## Ejercicio 3

Mejora el programa anterior para que muestre por separado la suma de los números pares y los impares.

Ejemplo 1:

- 1 Introduce el número de inicio: 4
- 2 Introduce el número de fin: 8
- 3 Los pares suman 18 y los impares 12

Ejemplo 2:

- 1 Introduce el número de inicio: 10
- 2 Introduce el número de fin: 15
- 3 Los pares suman 36 y los impares 39

## Ejercicio 4

Escribe un programa que solicite al usuario un nombre de usuario y contraseña. El programa mostrará el mensaje “¡Bienvenido!” si el usuario introduce los siguientes datos:

- Nombre de usuario: root
- Contraseña: toor

Si los datos de acceso son incorrectos mostrará el mensaje “Acceso fallido” y el programa finalizará.

Ejemplo 1:

- 1 Introduce el nombre de usuario: root
- 2 Introduce la contraseña: toor
- 3 ¡Bienvenido!

Ejemplo 2:

- 1 Introduce el nombre de usuario: root
- 2 Introduce la contraseña: 123456
- 3 Acceso fallido

## Ejercicio 5

Mejora el programa anterior para que solo permita 3 intentos. Cada vez que el usuario introduzca datos de acceso incorrectos el programa mostrará el mensaje: “Datos incorrectos. Le quedan X intentos.”, siendo X el número de intentos restantes. Tras el tercer fallo el programa mostrará el mensaje “Has agotado todos tus intentos.” y finalizará.

Ejemplo:

```
1 Introduce el nombre de usuario: root
2 Introduce la contraseña: 123456
3 Datos incorrectos. Le quedan 2 intentos.
4 Introduce el nombre de usuario: root
5 Introduce la contraseña: abcd
6 Datos incorrectos. Le quedan 1 intentos.
7 Introduce el nombre de usuario: root
8 Introduce la contraseña: 123abc
9 Datos incorrectos. Le quedan 0 intentos.
10
11 Has agotado todos tus intentos.
```

## Ejercicio 6

Crea un programa que reciba 5 números del usuario y muestre el mayor de todos por pantalla.

Ejemplo:

```
1 Introduce un número: 5
2 Introduce un número: -10
3 Introduce un número: 2
4 Introduce un número: 14
5 Introduce un número: 7
6 El número más alto es: 14
```

## Ejercicio 7

Mejora el programa anterior, de forma que el usuario pueda introducir tantos números como quiera. El programa solicitará números al usuario hasta que introduzca la palabra “fin”. Entonces mostrará el mayor de todos por pantalla.

Ejemplo:

```
1 Introduce un número: 6
2 Introduce un número: 9
3 Introduce un número: 11
4 Introduce un número: 3
5 Introduce un número: 5
6 Introduce un número: fin
7 El número más alto es: 11
```

## 6. Listas y tuplas

Las listas permiten **guardar más de un elemento** dentro de una variable, y además hacerlo en un orden concreto. Pueden contener un número **ilimitado** de elementos de **cualquier tipo**:

```
1     # Lista vacía
2     lista_vacia = []
3
4     # Lista con valores
5     alumnos = ["Ane", "Unai", "Itziar", "Mikel"]
6
7     # Acceder a elementos
8     print(alumnos[0]) # muestra "Ane"
9     print(alumnos[1]) # muestra "Unai"
10    print(alumnos[2]) # muestra "Itziar"
11    print(alumnos[-1]) # muestra "Mikel"
12
13    # Cambiar un elemento
14    alumnos[0] = "Nora"
```

Los **métodos más utilizados** con las listas son los siguientes:

Método	Acción
<code>alumnos.append("Amaia")</code>	Inserta "Jon" al final de la lista
<code>alumnos.insert(0, "Amaia")</code>	Inserta "Amaia" en la posición 0
<code>alumnos.remove("Amaia")</code>	Elimina la primera aparición de "Amaia" de la lista
<code>alumnos.pop()</code>	Elimina el último elemento de la lista
<code>alumnos.pop(3)</code>	Elimina el cuarto elemento de la lista
<code>alumnos.clear()</code>	Elimina todos los elementos de la lista
<code>alumnos.index("Amaia")</code>	Devuelve el índice de la primera aparición de "Amaia"
<code>alumnos.sort()</code>	Ordena la lista (los elementos deben ser comparables)
<code>sorted(alumnos)</code>	Devuelve una copia de la lista 'alumnos' ordenada (no ordena la pasada como parámetro)
<code>alumnos.reverse()</code>	Ordena la lista en orden inverso
<code>alumnos.copy()</code>	Devuelve una copia de la lista
<code>alumnos.extend(otra_lista)</code>	Fusiona las dos listas

## Acceder a varios elementos de una lista

Si queremos acceder a un subconjunto de elementos de la lista, es posible hacerlo de la siguiente manera:

```
1 lista = ['a', 'b', 'c', 'd', 'e', 'f']
2 # Elementos de la segunda a la cuarta posición
3 print(lista[1:3]) # Salida: ['b', 'c']
4
5 # Elementos desde la primera hasta la cuarta posición
6 print(lista[:3]) # Salida: ['a', 'b', 'c']
7
8 # Elementos desde la tercera posición hasta el final
9 print(lista[2:]) # Salida: ['c', 'd', 'e', 'f']
```

## Recorrer una lista

La forma habitual de recorrer una lista es mediante la sentencia `for`, tal y como muestra el ejemplo a continuación:

```
1     for elemento in ['Python', 'JavaScript', 'JAVA']:
2         print("Programo en ", elemento)
```

De igual manera se podría hacer mediante la sentencia `while`:

```
1     lista = ['Python', 'JavaScript', 'JAVA']
2     i = 0
3     sizeofList = len(lista)
4     while i < sizeofList :
5         print(lista[i])
6         i += 1
```

## Tuplas

Las tuplas son **listas inmutables**. Es decir, una vez declaradas, no se pueden realizar modificaciones sobre ellas (añadir/eliminar elementos o hacer modificaciones sobre ellos). Para definir una tupla se escriben los elementos entre paréntesis:

```
1 valores = (1,2,3,4,5)
2 print(valores) # Salida: (1, 2, 3, 4, 5)
3
4 # tuple with mixed datatypes
5 valores_mixtos = (1, "Hola", 2.5, False)
6 print(valores_mixtos) # Salida: (1, 'Hola', 2.5, False)
```

El acceso a sus elementos se hace de igual que con listas:

```
1 valores = ("a", "b", "c", "d", "e", "f")
2 print(valores[1]) # Salida: 'b'
3 print(valores[2:4]) # Salida: ('c', 'd')
```

Una acción típica de las tuplas es “desempaquetar” (unpack) sus valores, es decir, asignarlos a variables directamente:

```
1 tupla = (1, "Hola", 2.5) # Creamos la tupla
2
3 var1, var2, var3 = tupla # Hacemos el unpack
4
5 print(var1) # 1
6 print(var2) # 'Hola'
7 print(var3) # 2.5
```

## Coding time!

### Ejercicio 1

Dada la siguiente lista [12, 23, 5, 29, 92, 64] realiza las siguientes operaciones y vete mostrando la lista resultante por pantalla:

1. Elimina el último número y añádelo al principio.
2. Mueve el segundo elemento a la última posición.
3. Añade el número 14 al comienzo de la lista.
4. Suma todos los números de la lista y añade el resultado al final de la lista.
5. Fusiona la lista actual con la siguiente: [4, 11, 32]
6. Elimina todos los números impares de la lista.
7. Ordena los números de la lista de forma ascendente.
8. Vacía la lista.

Resultado:

```
1 [64, 12, 23, 5, 29, 92]
2 [64, 23, 5, 29, 92, 12]
3 [14, 64, 23, 5, 29, 92, 12]
4 [14, 64, 23, 5, 29, 92, 12, 239]
5 [14, 64, 23, 5, 29, 92, 12, 239, 4, 11, 32]
6 [14, 64, 92, 12, 4, 32]
7 [4, 12, 14, 32, 64, 92]
8 []
```

## Ejercicio 2

Crea un programa que solicite al usuario 5 números y los guarde en una lista. A continuación el programa pedirá otros 5 números al usuario almacenándolos en una segunda lista. El programa mostrará al usuario una lista que contenga los números que tienen en común las dos listas anteriores.

- Ejemplo: Lista 1 = [6, 14, 11, 78, 5] y Lista 2 = [3, 14, 22, 78, 9]
- Resultado: [14, 78]

# 7. Diccionarios

Un diccionario es un conjunto de parejas **clave- valor** (key-value). Es decir, se accede a cada elemento a partir de su clave. Se definen de la siguiente manera:

```
1 estudiante = {
2     "nombre": "Iñaki Perurena",
3     "edad": 30,
4     "nota_media": 7.25,
5     "repetidor" : False
6 }
```

Las **claves tienen que ser únicas** y estar formadas por un **string o un número**. Para acceder al valor de una clave existen dos maneras distintas:

```
1 # Acceder al valor de una clave
2 edad = estudiante["edad"] # devuelve el valor de 'edad'
3 nota_media = estudiante.get("nota_media") # devuelve el valor de 'nota_media'
4
5 # Insertar o actualizar un valor:
6 estudiante["edad"] = 25 # actualiza el valor de 'edad'
7 estudiante["suspensos"] = 3 # inserta una nueva pareja clave - valor
8
9 # insertar una pareja clave - valor o actualizar si ya existe:
10 estudiante.update({'aprobados': '8'})
```

Algunos de los métodos más utilizados son los siguientes:

Método	Acción
diccionario.keys()	Devuelve todas las claves del diccionario
diccionario.values()	Devuelve todos los valores del diccionario
diccionario.pop(clave[, <default>])	Elimina la clave del diccionario y devuelve su valor asociado. Si no la encuentra y se indica un valor por defecto, devuelve el valor por defecto indicado.
diccionario.clear()	Vacía el diccionario
clave in diccionario	Devuelve True si el diccionario contiene la clave o False en caso contrario.
valor in diccionario.values()	Devuelve True si el diccionario contiene el valor o False en caso contrario.

## Recorrer un diccionario

La forma más habitual de recorrer un diccionario es mediante la sentencia `for`. Al recorrer un diccionario, por defecto se iterará sobre sus claves:

```
1 diccionario = {'a':1, 'b':2, 'c':3}
2 for key in diccionario:
3     print(key)
4
5 # Resultado: a b c
```

Es decir, el código anterior será equivalente al siguiente:

```
1 diccionario = {'a':1, 'b':2, 'c':3}
2 for key in diccionario.keys():
3     print key
4
5 # Resultado: a b c
```

Por lo tanto, para iterar accediendo a los valores, realizaremos lo siguiente:

```
1 diccionario = {'a':1, 'b':2, 'c':3}
2 for key in diccionario:
3     print(diccionario[key])
4
5 # Resultado: 1 2 3
```

Otra manera alternativa sería empleando la función `items()`, la cual devuelve el diccionario como tuplas de tipo `(key,value)`:

```
1 diccionario = {'a':1, 'b':2, 'c':3}
2 for key, value in diccionario.items():
3     print("El valor de %s is %d" % (key, value))
4
5 # Resultado:
6 # El valor de a is 1
7 # El valor de b is 2
8 # El valor de c is 3
```

## Borrar un elemento

Para borrar un elemento de un diccionario se utiliza la instrucción `del`.

```
1 edades = {
2     "Ane" : 22,
3     "Jokin" : 27,
4     "Aitor" : 15
5 }
6 del edades["Aitor"]
```

Otra alternativa también utilizada y mencionada anteriormente es la función `pop()`, el cual devuelve el valor del elemento eliminado:

```
1 edades = {
2     "Ane" : 22,
3     "Jokin" : 27,
4     "Aitor" : 15
5 }
6 edades.pop("Aitor")
```

Un diccionario nunca debería contener dos claves iguales. No obstante, en caso de contener una clave repetida, tanto `del` como `pop()` eliminarán todas las claves coincidentes.

## Coding time!

### Ejercicio 1

Crea un programa que recorra una lista y cree un diccionario que contenga el número de veces que aparece cada número en la lista.

- Ejemplo: [12, 23, 5, 12, 92, 5, 12, 5, 29, 92, 64, 23]
- Resultado: {12: 3, 23: 2, 5: 3, 92: 2, 29: 1, 64: 1}

### Ejercicio 2

Recorre un diccionario y crea una lista solo con los valores que contiene, sin añadir valores duplicados.

- Ejemplo: {'Mikel': 3, 'Ane': 8, 'Amaia': 12, 'Unai': 5, 'Jon': 8, 'Ainhoa': 7, 'Maite': 5}
- Resultado: [3, 8, 12, 5, 7]

### Ejercicio 3

Crea un programa de Login que compruebe el usuario y contraseña en el diccionario a continuación:

```
1 usuarios = {
2     "iperurena": {
3         "nombre": "Iñaki",
4         "apellido": "Perurena",
5         "password": "123123"
6     },
7     "fmuguruza": {
8         "nombre": "Fermín",
9         "apellido": "Muguruza",
10        "password": "654321"
11    },
12    "aolaizola": {
13        "nombre": "Aimar",
14        "apellido": "Olaizola",
15        "password": "123456"
16    }
17 }
```

El usuario tendrá un máximo de 3 intentos, y al acceder correctamente se mostrará el nombre y apellido del usuario.

## Ejercicio 4

Crear un programa que permita introducir a un profesor las notas de sus estudiantes (máximo 10 estudiantes). Los datos se deberán almacenar en un diccionario como el siguiente:

```
1 estudiantes = {
2     "1": {
3         "nombre": "Lorea",
4         "nota": 8
5     },
6     "2": {
7         "nombre": "Markel",
8         "nota": "4.2"
9     },
10    "3": {
11        "nombre": "Julen",
12        "nota": 6.5
13    }
14 }
```

Una vez introducidos todos los datos, el programa mostrará una lista con los nombres de los estudiantes que han suspendido y otra con los que han aprobado. También calculará y mostrará la nota media de la clase.

# 8. Funciones

Una función es un grupo de sentencias que realizan una tarea concreta. Esta forma de agrupar código es una forma de **ordenar** nuestra aplicación en pequeños bloques, **facilitando así su lectura** y permitiendo **reutilizar** el código que contienen sin esfuerzo.

## Definir y llamar a una función

La sintaxis de una función en Python es la siguiente:

```
1 def saludo(nombre):
2     # código de la función
3     print("Hola, " + nombre+ ". ¡Bienvenido!")
```

Se escribe la palabra reservada `def` seguida del nombre de la función y sus parámetros entre paréntesis.

Para **llamar a una función** solo hay que escribir el nombre de la función seguida de los parámetros (si los hubiera) entre paréntesis.

```
1 >>> saludo('Maitane')
2 Hola, Maitane. ¡Bienvenida!
```

Es posible asignar al parámetro un **valor por defecto**.

```
1 def saludo(nombre = "Anónimo"):
2     print("Hola, " + nombre+ ". ¡Bienvenido!")
3
4 saludo("Leire") # Hola, Maitane. ¡Bienvenida!
5 saludo() # Hola, Anónimo. ¡Bienvenida!
```

Existen dos tipos de parámetros o argumentos:

- **Parámetros posicionales:** la posición en la que se pasan importa
- **Parámetros con palabra clave (keyword arguments):** la posición no importa, se indica una clave para cada parámetro.

```
1 def suma(a, b):
2     resultado = a + b
3     print(resultado)
4 suma(45, 20) # parámetros posicionales
5 suma(b=20,a=45) # parametros mediante clave
```

Las funciones pueden **devolver un valor** utilizando la palabra `return`. Una vez devuelto un valor, la función finaliza su ejecución.

```
1 def suma(a, b):
2     resultado = a + b
3     return resultado
4
5 print(suma(4,5)) # 9
```

## Funciones con argumentos múltiples

Es posible recibir un **número desconocido** de parámetros añadiendo un `*` en la definición de la función.

```
1 def suma_todo(*args):
2     resultado = 0
3     for i in args:
4         resultado += i
5     return resultado
6 v, w, x, y, z = 5, 2, 12, 6, 9
7 total = suma_todo(v, w, x, y, z)
8 print("La suma total es:" + str(total)) # La suma total es: 34
```

## Ámbito de las variables (scope)

El ámbito de una variable (*scope*) se refiere a la zona del programa dónde una variable “existe”. Fuera del ámbito de una variable no podremos acceder a su valor ni manejarla.

Los parámetros y variables definidos en una función no estarán accesibles fuera de la función. A este ámbito se le conoce como **ámbito local**. Es importante mencionar que una vez ejecutada una función, el valor de las variables locales no se almacena, por lo que la próxima vez que se llame a la función, ésta no recordará ningún valor de llamadas anteriores.

```
1 def calcula():
2     a = 1
3     print("Dentro de la función:", a)
4
5 a = 5
6 calcula()
7 print("Fuera de la función:", a)
8
9 ### Output ###
10 # Dentro de la función:1
11 # Fuera de la función:5
```

Por el contrario, las variables definidas fuera de una función sí que están accesibles desde dentro de la función. Se considera que están en el **ámbito global**. No obstante, no se podrán modificar dentro de la función a no ser que estén definidas con la palabra clave `global`.

## Coding time!

### Ejercicio 1

Crea un programa que determine si un número es primo o no. Deberás crear la función `esPrimo()` que reciba como parámetro un número y devuelva `True` o `False` indicando si el número es primo o no.

### Ejercicio 2

Crea un programa que genere un número aleatorio del 1 al 10. El usuario tendrá que adivinarlo, y el programa tras cada intento le indicará al usuario si el número es más alto, bajo o si ha acertado. La lógica para dar la respuesta al usuario deberá estar incluida en una función a la que se llamará tras cada intento.

Nota: Para la creación del número aleatorio, utiliza el siguiente código:

```
1 from random import randint, uniform, random
2
3 numero = randint(0,10)
```

### Ejercicio 3

Crea un programa que reciba un número del 1 al 20 introducido por el usuario y compruebe si está dentro de la siguiente lista: `[6, 14, 11, 3, 2, 1, 15, 19]`. Implementa una función que se asegure que el número introducido por el usuario está en el rango indicado y otra que compruebe si está dentro de la lista. Trata de crear las funciones de forma que puedan ser reutilizadas lo máximo posible en otros programas.

# 9. Excepciones

Las excepciones son **errores en la ejecución de un programa** que hacen que el programa termine de forma inesperada. Normalmente ocurren debido a un uso indebido de los datos (p.ej. una división entre cero). La manera de controlar las excepciones es agrupando el código en 2 bloques (más 1 opcional):

- `try`: agrupa el bloque de código en el que se pueda dar una excepción.
- `catch`: contiene el código a ejecutar en caso de que la excepción haya sido lanzada.
- `finally` (opcional): permite ejecutar un bloque de código siempre, se haya capturado o no una excepción.

```
1 try:
2     numero = int(input('Introduce un número: '))
3     dividendo = 150
4     resultado = dividendo / numero
5     print(resultado)
6 except ValueError:
7     print('Número inválido')
8 except ZeroDivisionError:
9     print('No se puede dividir entre 0')
10 finally:
11     print("Ejecutando finally antes de salir")
```

También es posible lanzar excepciones de forma controlada mediante la sentencia `raise`.

```
1 raise NameError('¡Soy una excepción!')
```

## Excepciones comunes

Hay algunas excepciones que son bastante comunes a la hora de programar en Python y que deberíamos contemplar en nuestros programas:

`TypeError` es lanzado cuando se intenta realizar una operación o una función sobre un objeto de un tipo inapropiado.

```
1 >>> '1'+1
2 Traceback (most recent call last):
3 File "<pysHELL#23>", line 1, in <module>
4 '1'+2
5 TypeError: must be str, not int
```

**ValueError** es lanzado cuando el argumento de una función es de un tipo inapropiado.

```
1 >>> int('hola')
2 Traceback (most recent call last):
3 File "<pysHELL#14>", line 1, in <module>
4 int('xyz')
5 ValueError: invalid literal for int() with base 10: 'hola'
```

**NameError** es lanzado cuando no utiliza un objeto que no existe.

```
1 >>> persona
2 Traceback (most recent call last):
3 File "<pysHELL#6>", line 1, in <module>
4 age
5 NameError: name 'persona' is not defined
```

**IndexError** es lanzado al intentar acceder a un índice que no existe en un array.

```
1 >>> lista = [1,2,3]
2 >>> lista[5]
3 Traceback (most recent call last):
4 File "<pysHELL#18>", line 1, in <module>
5 lista[5]
6 IndexError: list index out of range
```

**KeyError** es lanzado cuando no se encuentra la clave (key),

```
1 >>> diccionario={'1':"esto", '2':"es", '3':"python"}
2 >>> diccionario['4']
3 Traceback (most recent call last):
4 File "<pysHELL#15>", line 1, in <module>
5 diccionario['4']
6 KeyError: '4'
```

**ModuleNotFoundError** es lanzado cuando no se encuentra el módulo indicado.

```
1 >>> import mimodulo
2 Traceback (most recent call last):
3 File "<pyshell#10>", line 1, in <module>
4 import mimodulo
5 ModuleNotFoundError: No module named 'mimodulo'
```

## Coding time!

### Ejercicio 1

Crea un programa que acceda a la posición que el usuario indique de la siguiente lista: [6, 14, 11, 3, 2, 1, 15, 19]. No olvides capturar las excepciones que puedan surgir en caso de que el usuario introduzca un índice incorrecto o que no exista en la lista.

### Ejercicio 2

Crea una aplicación reciba la clave de un diccionario y acceda a uno de sus valores. Asegúrate de que capturas las excepciones que puedan saltar al intentar acceder a una clave del diccionario inexistente.

# 10. Clases y objetos

Python soporta la programación orientada a objetos. Esto quiere decir que podemos definir entidades agrupando (encapsulando) sus atributos y comportamiento (métodos) en clases.

La definición de una clase en Python se hace de la siguiente manera:

```
1 class Persona:
2     # atributos
3     nombre = "Josune"
4     edad = 24
5
6     # metodos
7     def camina(self):
8         print(self.nombre + " está caminando")
```

Una clase es como una plantilla o modelo para crear a partir de ella objetos. Esta plantilla contiene la información para definir cómo serán los objetos, es decir, qué atributos y métodos tendrán.

A partir de una clase se pueden crear tantos **objetos** como se desee. Los objetos de una clase se conocen como **instancias**. Cada objeto **contiene los atributos y métodos de la clase** y podrá asignar a esos atributos unos valores concretos. Esto se conoce como el **estado de un objeto**.

```
1 p1 = Persona() # la variable p1 contiene un objeto de la clase Persona
2 p1.camina()
3 print(p1.nombre)
4 print(p1.edad)
```

Una función dentro de una clase se conoce como **método**. Las clases contienen el método especial `__init__` conocido como **constructor** y que sirve para inicializar un objeto. Al crear un objeto siempre se llama al constructor. Una diferencia importante con otros lenguajes como Java es que solo se puede definir un único constructor.

```
1 class Persona:
2     def __init__(self, nombre, apellidos, edad):
3         self.nombre= nombre
4         self.apellidos = apellidos
5         self.edad = edad
6
7     def camina(self):
8         print(self.nombre + " está caminando")
```

En la creación del objeto es necesario indicar los argumentos del constructor:

```
1 p1 = Persona("Mike", "Mendiola", 25)
2 p1.camina()
3 print(p1.nombre)
4 print(p1.edad)
```

El parámetro `self` de los métodos es una referencia a la propia instancia y se utiliza para acceder a las variables que pertenecen a la clase. Si no se define un constructor, la clase hereda uno que únicamente recibe el argumento `self`.

## Atributos de clase vs Atributos de instancia

Los atributos definidos dentro del constructor se conocen como **atributos de instancia**, por lo tanto, los atributos definidos dentro de la clase pero fuera del constructor se conocen como **atributos de clase**.

La principal diferencia es que un atributo de clase puede ser accedido aunque no existan instancias de la clase. Además, si se modifica su valor, se modificará el valor en todas las instancias existentes de dicha clase.

```
1 class Demo:
2     atrib_estatico = 123 # compartido por todos los objetos
3     def __init__(self, numero):
4         self.atrib_instancia = numero # específico de cada objeto
5
6 c1 = Demo(456)
7 c2 = Demo(789)
8
9 # Valor inicial
10 print(f"C1: Estatico {1.atrib_estatico} - Instancia: {c1.atrib_instancia}")
11 # output: C1: Estatico 123 - Instancia: 456
12 print(f"C2: Estatico {c2.atrib_estatico} - Instancia: {c2.atrib_instancia}")
```

```

13 # output: C2: Estatico 123 - Instancia: 789
14
15 Demo.atrib_estatico = -1
16
17 print(f"C2: Estatico {c2.atrib_estatico} - Instancia: {c2.atrib_instancia}")
18 # output: C2: Estatico -1 - Instancia: 456
19 print(f"C2: Estatico {c2.atrib_estatico} - Instancia: {c2.atrib_instancia}")
20 # output: C2: Estatico -1 - Instancia: 789
21
22 c1.atrib_instancia = 999
23
24 print(f"C1: Estatico {c1.atrib_estatico} - Instancia: {c1.atrib_instancia}")
25 # output: C1: Estatico -1 - Instancia: 999
26
27 print(f"C2: Estatico {c2.atrib_estatico} - Instancia: {c2.atrib_instancia}")
28 # output: C2: Estatico -1 - Instancia: 789

```

Es importante remarcar que para acceder a los atributos de instancia se debe utilizar la palabra reservada `self`, la cual hace referencia al objeto actual. En Python no podemos utilizar `self` en cualquier momento, para utilizarlo hay que indicarlo en los métodos cómo el primer parámetro recibido.

```

1 class Persona:
2     def __init__(self, nombre, apellidos, edad):
3         self.nombre= nombre
4         self.apellidos = apellidos
5         self.edad = edad
6
7     def camina(self): # es necesario indicar 'self' como primer argumento
8         print(self.nombre + " está caminando")
9
10 p1 = Persona("Mike", "Mendiola", 25) # no hay que pasarle 'self'
11 p1.camina() # no hay que pasarle 'self'
12 print(p1.nombre)
13 print(p1.edad)

```

## Private y protected

A diferencia de otros lenguajes de Programación Orientada a Objetos, todos los métodos y atributos en Python son públicos. Es decir, **no es posible definir una variable como** `private` o `protected`.

Existe una convención de añadir como prefijo un **guión bajo** (`_`) a los atributos que consideramos como **protected** y dos guiones bajos (`__`) a las variables que consideramos **private**.

```
1 class Persona:
2     def __init__(self, nombre, edad):
3         self._nombre = nombre # atributo protected
4         self.__edad = edad # atributo private
```

## Coding time!

### Ejercicio 1

Crea la clase Coche que contenga las siguientes propiedades:

- matrícula (string)
- marca (string)
- kilometros\_recorridos (float)
- gasolina (float)

La clase tendrá un método llamado `avanzar()` que recibirá como argumento el número de kilómetros a conducir y sumará los kilómetros recorridos al valor de la propiedad `kilometros_recorridos`. El método también restará al valor de `gasolina` el resultado de los kilómetros multiplicado por 0'1.

La clase también contendrá otro método llamado `repostar()` que recibirá como argumento los litros introducidos que deberán sumarse a la variable `gasolina`.

Por último, será necesario controlar que el método `avanzar` nunca obtendrá un número negativo en la gasolina. En dicho caso, deberá mostrar el siguiente mensaje: “Es necesario repostar para recorrer la cantidad indicada de kilómetros”.

Ejemplo:

```
- avanzar(50) # gasolina = 50
- avanzar(100) # kilometros_recorridos = 100, gasolina = 40
- avanzar(40) # kilometros_recorridos = 140, gasolina = 36
- avanzar(180) # kilometros_recorridos = 320, gasolina = 18
```

### Ejercicio 2

Crea una clase Robot que simule los movimientos de un robot y calcule la posición en la que se encuentra cada momento. El robot se moverá por un tablero infinito de coordenadas X e Y, podrá realizar los siguientes movimientos:

- Avanzar hacia adelante (A)
- Retroceder (R)
- Avanzar hacia la izquierda (I) o hacia la derecha (D)

El robot tendrá un método llamado `mueve()` que recibirá la orden como parámetro y otro método, `posicion_actual()`, que indicará su posición en las coordenadas X e Y. Al crear el robot este se inicializará a las coordenadas (0,0).

Puedes utilizar el siguiente código para probar la clase creada:

```
1 miRobot = Robot()
2 orden = "A"
3 while orden != 'fin':
4     orden = input("Introduce la orden: ")
5     miRobot.mueve(orden)
6     print(miRobot.posicion_actual())
```

Ejemplo:

```
1 >> Introduce la orden: A
2 Posición actual: 1,0
3 >> Introduce la orden: A
4 Posición actual: 2,0
5 >> Introduce la orden: I
6 Posición actual: 2,-1
7 >> Introduce la orden: A
8 Posición actual: 3,-1
9 >> Introduce la orden: I
10 Posición actual: 3,-2
11 >> Introduce la orden: D
12 Posición actual: 3,-1
13 >> Introduce la orden: R
14 Posición actual: 2,-1
15 >> Introduce la orden: fin
```

## Ejercicio 3

Mejora el ejercicio anterior de forma que el robot pueda recibir una secuencia de movimientos. Por ejemplo:

- `mueve("AADDADIR")`

También deberá tener otros dos métodos: uno que devuelva todas las órdenes recibidas y otro capaz de listar los movimientos necesarios para volver a la posición inicial (0,0).

Aquí tienes un ejemplo de una posible ejecución del programa:

```
1 Introduce la orden: AADAD
2 Posición actual: 3,2
3 Introduce la orden: IAADR
4 Posición actual: 4,2
5 Introduce la orden: fin
6 Posición actual: 4,2
7
8 Órdenes recibidas: AADADIAADRfin
9 Secuencia para posición inicial: RRRRII
```

## Ejercicio 4

Crea la clase Triangulo que almacene la longitud de cada uno de sus lados. Deberá contener los siguientes métodos:

- `area()`: devuelve el área del triángulo
- `forma()`: indica si se trata de un triángulo equilátero, isósceles o irregular.
- `perímetro()`: devuelve el perímetro del triángulo.

# 11. Herencia

La herencia es una técnica de la Programación Orientada a Objetos en la que una clase (conocida como **clase hija** o **subclase**) hereda todos los métodos y propiedades de otra clase (conocida como **padre** o **clase base**).

La sintaxis para definir una clase que herede de otra es la siguiente:

```
1 class ClaseBase:
2     # código de la clase base
3
4 class Subclase(BaseClass):
5     # código de la subclase
```

La subclase puede añadir funcionalidades. Esta técnica permite **reutilizar código**.

```
1 class Dispositivo:
2     def __init__(self, identificador, marca):
3         self.identificador = identificador
4         self.marca = marca
5
6     def conectar(self):
7         print(" ¡Conectado!")
8
9     # la clase base se indica entre paréntesis
10 class Teclado(Dispositivo):
11     def __init__(self, identificador, marca, tipo):
12         # llamada al constructor del padre
13         Dispositivo.__init__(self, identificador, marca)
14         self.tipo = tipo
15     # metodo de la subclase
16     def pulsar_tecla(self, tecla):
17         print(tecla)
18
19 t1 = Teclado("0001", "Logitech", "AZERTY")
20 print(f"Id: {t1.identificador}, Marca: {t1.marca}, tipo: {t1.tipo}")
21 t1.conectar()
22 t1.pulsar_tecla("a")
```

## Herencia múltiple

Python soporta la herencia múltiple, es decir, hereda métodos y atributos de más de un padre. En caso de heredar atributos o métodos con el mismo nombre, Python dará prioridad al posicionado más a la izquierda en la declaración.

```
1 # en caso de conflicto Dispositivo tendrá prioridad sobre Periférico
2 class Teclado(Dispositivo, Periférico):
3     # cuerpo de la clase
```

## Coding time!

### Ejercicio 1

Continuando con el Ejercicio 1 del tema anterior, crea una clase Vehículo y otra llamada Bicicleta. La clase Vehículo será el padre de Coche y Bicicleta y contendrá las propiedades y/o métodos comunes de ambos. La bicicleta no tendrá gasolina ni repostará, pero cada 50 kilómetros necesitará invocar al método `hinchar_ruedas()` o no podrá continuar.

Puedes utilizar este código para comprobar que todo funciona correctamente:

```
1 # Coche:
2 coche = Coche("1122PKL", "Audi")
3 coche.repostar(20)
4 coche.avanzar(120)
5 print(f"Total de kms coche: {coche.kilometros}. Gasolina: {coche.gasolina}")
6 coche.avanzar(40)
7 print(f"Total de kms coche: {coche.kilometros}. Gasolina: {coche.gasolina}")
8
9 # Bicicleta
10 bicicleta = Bicicleta("BH")
11 bicicleta.avanzar(30)
12 print(f"Total de kms bici: {bicicleta.kilometros}")
13 bicicleta.avanzar(25)
14 print(f"Total de kms bici: {bicicleta.kilometros}")
15 bicicleta.hinchar_ruedas()
16 bicicleta.avanzar(25)
17 print(f"Total de kms bici: {bicicleta.kilometros}")
```

Resultado:

```
1 Total de kms coche: 120. Gasolina: 8.0
2 Total de kms coche: 160. Gasolina: 4.0
3
4 Total de kms bici: 30
5 Es necesario hinchar para recorrer la cantidad indicada de kms.
6 Total de kms bici: 30
7 Total de kms bici: 55
```

## Ejercicio 2

Continuando con el Ejercicio 4 del tema anterior, crea una clase `Poligono` y otra llamada `Cuadrado`. La clase `Poligono` será el padre de `Triangulo` y `Cuadrado`, y contendrá las propiedades y métodos comunes de ambos. Ambos tendrán también otra propiedad llamada `color`.

```
1 t1 = Triangulo("rojo",[2, 5, 2])
2 print(f"Es un {t1.forma()} {t1.color} con {t1.perimetro()}m de perímetro.")
3
4 c1 = Cuadrado("azul",[4, 4, 4, 4])
5 print(f" Cuadrado {c1.color} con {c1.perimetro()}m de perímetro.")
```

Resultado:

```
1 Es un Triángulo isósceles rojo con 9m de perímetro.
2 Cuadrado azul con 16m de perímetro.
```

# 12. Módulos y Paquetes

## Módulos

Un módulo es un archivo de Python que contiene variables, funciones y clases. Es una forma de ordenar y reutilizar código ya que todo el contenido de un módulo es accesible por los archivos que lo importen.

```
1 # mundo.py
2
3 def hola_mundo():
4     print("¡Hola Mundo!")
5
6 def adios_mundo():
7     print("¡Adios Mundo!")
```

Para acceder a las funciones desde otro archivo Python se utiliza la palabra reservada `import`:

```
1 # app.py
2
3 import mundo
4
5 # Llamada a la función
6 mundo.hola_mundo()
```

También existe la posibilidad de importar únicamente objetos concretos de un módulo mediante la sintaxis `from ... import`:

```
1 # app.py
2
3 from mundo import adios_mundo
4
5 # Llamada a la función
6 adios_mundo()
```

De esta forma no es necesario escribir el nombre del módulo antes de utilizar la función. De igual manera, se pueden importar varios objetos de un módulo separándolos por una coma:

```
1 # app.py
2
3 from mundo import adios_mundo, hola_mundo
```

Para importar todos los los objetos de un módulo basta con utilizar el asterisco:

```
1 # app.py
2
3 from mundo import *
```

## Localización de los módulos

Al importar un módulo Python lo buscara en los siguientes directorios:

1. En el directorio actual.
2. En los directorios declarados en el PYTHONPATH (variable de entorno que contiene un listado de directorios)
3. En el directorio de instalación de Python por defecto (en UNIX normalmente `‘/usr/local/lib/python/’`)

## Paquetes

Es posible agrupar los módulos que tienen relación en un mismo directorio. Estos directorios son conocidos en Python como paquetes y deben contener siempre un archivo llamado `__init__.py` para que Python lo reconozca como un paquete.

A medida que desarrollamos una aplicación es habitual agrupar los archivos en directorios (paquetes) para tener el código organizado.

Para cargar un módulo ubicado en un paquete lo haremos de la siguiente forma:

```
1 import mipaquete.mundo
```

o bien de la siguiente manera:

```
1 from mipaquete import mundo
```

También es posible importar elementos concretos de un módulo:

```
1 from mipaquete.mundo import adios_mundo, hola_mundo
```

# 13. Próximos pasos

Como ya te habíamos adelantado, **tu viaje en el mundo de Python solo acaba de empezar**. Python es uno de los lenguajes de programación con mayor potencial, y ahora que ya dispones de una sólida base de los fundamentos básicos, podrás comenzar a explorar otros mundos como:

- Desarrollo de aplicaciones web
- Internet of Things
- Análisis de datos (Data Science, Machine Learning, Big Data,...)
- Scripting
- Aplicaciones embebidas
- Desarrollo de aplicaciones de escritorio

Podrás profundizar en cada una de estas ramas a partir de los conocimientos de los que dispones. En función del camino que elijas, podrás aprender frameworks como Django, Flask, Numpy, PyTorch, PANDAS, TensorFlow, Libmraa, ...

A continuación te listo algunos de los recursos a los que sin duda alguna merecen la pena darles un vistazo:

- [Tutoriales de Data Camp](https://www.learnpython.org/)<sup>1</sup>
- [Head First Python](https://www.amazon.es/dp/1491919531/?tag=devdetailpa03-21)<sup>2</sup>
- [Python Crash Course](https://www.amazon.es/Python-Crash-Course-Hands-Project-Based-ebook/dp/B018UXJ9RI/ref=sr_1_3?__mk_es_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180835&sr=8-3)<sup>3</sup>
- [Learn Python 3 the Hard Way](https://www.amazon.es/Learn-Python-Hard-Way-Introduction-ebook/dp/B07378P8W6/ref=sr_1_7?__mk_es_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180896&sr=8-7)<sup>4</sup>
- [Python for Data Analysis](https://www.amazon.es/Python-Data-Analysis-Wrangling-IPython-ebook/dp/B075X4LT6K/ref=sr_1_13?__mk_es_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180896&sr=8-13)<sup>5</sup>
- [Google's Python Class](https://developers.google.com/edu/python/)<sup>6</sup>
- [Curso "Introduction to Python", de Udacity](https://www.udacity.com/course/introduction-to-python--ud1110)<sup>7</sup>

Recuerda, lo único que necesitas saber es que **tú puedes aprender cualquier cosa**.

---

<sup>1</sup><https://www.learnpython.org/>

<sup>2</sup><https://www.amazon.es/dp/1491919531/?tag=devdetailpa03-21>

<sup>3</sup>[https://www.amazon.es/Python-Crash-Course-Hands-Project-Based-ebook/dp/B018UXJ9RI/ref=sr\\_1\\_3?\\_\\_mk\\_es\\_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180835&sr=8-3](https://www.amazon.es/Python-Crash-Course-Hands-Project-Based-ebook/dp/B018UXJ9RI/ref=sr_1_3?__mk_es_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180835&sr=8-3)

<sup>4</sup>[https://www.amazon.es/Learn-Python-Hard-Way-Introduction-ebook/dp/B07378P8W6/ref=sr\\_1\\_7?\\_\\_mk\\_es\\_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180896&sr=8-7](https://www.amazon.es/Learn-Python-Hard-Way-Introduction-ebook/dp/B07378P8W6/ref=sr_1_7?__mk_es_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180896&sr=8-7)

<sup>5</sup>[https://www.amazon.es/Python-Data-Analysis-Wrangling-IPython-ebook/dp/B075X4LT6K/ref=sr\\_1\\_13?\\_\\_mk\\_es\\_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180896&sr=8-13](https://www.amazon.es/Python-Data-Analysis-Wrangling-IPython-ebook/dp/B075X4LT6K/ref=sr_1_13?__mk_es_ES=%C3%85M%C3%85C5%BD%C3%95%C3%91&keywords=python&qid=1570180896&sr=8-13)

<sup>6</sup><https://developers.google.com/edu/python/>

<sup>7</sup><https://www.udacity.com/course/introduction-to-python--ud1110>