

Índice:

1. Introducción a Python.....	7
1.1 Características del lenguaje.....	7
1.2 Ventajas.....	7
1.3 Desventajas	7
1.4 Uso de Python.	8
1.5 Evolución de Python.....	8
2. Instalación de Python y otros programas. Preparación del entorno de trabajo.....	10
2.1 Probar el intérprete.....	10
2.2 Probar el editor.	10
2.3 Probar el depurador.....	11
2.4 Instalación de Python.....	11
2.5 Gestión de paquetes.	13
2.6 Entornos virtuales.	13
3. Tipos de datos básicos en Python.....	15
3.1 Variables.....	15
Convenciones para nombres:.....	16
Constantes.....	16
3.2 Asignación de valores a variables.....	16
Conocer el valor de una variable.....	17
Conocer el tipo de una variable.	17
3.3 Números.....	18
3.3.1 Booleanos.....	18
3.3.2 Enteros.	18
Operaciones con enteros.	19
Asignación aumentada.....	19
Valor absoluto	20
3.3.3 Flotantes.....	20
3.3.4 Conversión de tipos.....	20
Conversión implícita.....	20
Conversión explícita	21
3.3.5 Bases numéricas.....	21
Base binaria.	21
Base octal.	22

Base hexadecimal.....	22
3.4 Cadenas de texto.....	22
3.4.1 Introducción a las cadenas de texto.....	22
Secuencias de escape.....	23
Uso avanzado de print().	24
Leer datos desde teclado.	24
3.4.2 Operaciones con cadenas de texto.	25
Combinar cadenas.....	25
Repetir cadenas.....	25
Obtener un carácter.....	25
Trocear una cadena.....	25
Longitud de una cadena.	26
Pertenencia de un elemento.....	26
Dividir una cadena.....	26
Limpiar cadenas.....	27
Realizar búsquedas.....	27
Reemplazar elementos.....	28
Mayúsculas y minúsculas.	28
Identificando caracteres.....	29
Interpolación de cadenas.	29
Formateando cadenas.....	30
Modo debug.	31
Casos de uso - dir().	31
4. Colecciones en Python. Estructuras de datos.	32
4.1 Listas.....	32
4.1.1 Operaciones con listas.	32
Obtener un elemento.....	32
Trocear una lista.....	33
Invertir una lista.	33
Añadir al final de la lista.	33
Añadir en cualquier posición de una lista.	34
Repetir elementos.....	34
Combinar listas.....	34
Modificar una lista.....	34
Borrar elementos.	35
Borrado completo de la lista.	36

Encontrar un elemento.	36
Pertenencia de un elemento.....	36
Número de ocurrencias.....	37
Convertir lista a cadena de texto.	37
Ordenar una lista.....	37
Longitud de una lista.	37
Iterar sobre una lista.	38
Iterar usando enumeración.....	38
Iterar sobre múltiples listas.....	38
Lista de argumentos de un programa (sys.argv).....	39
Funciones matemáticas.....	39
Listas de listas.....	39
4.2 Tuplas.	40
4.2.1 Creación de tuplas.....	40
4.2.2 Modificación de tuplas.....	41
4.2.3 Conversión a tuplas.	41
4.2.4 Operaciones con tuplas.....	41
4.2.5 Desempaquetado de tuplas.	41
Intercambio de valores.....	42
Desempaquetado extendido.....	42
Desempaquetado genérico.	42
4.2.6 Tuplas vs Listas.	43
4.3 Diccionarios.	43
4.3.1 Creación de diccionarios.	43
4.3.2 Conversión a diccionarios.	44
Diccionario vacío.	44
Creación con dict().	44
4.3.3 Operaciones con diccionarios.	45
Obtener un elemento.....	45
Usando get().	45
Añadir o modificar un elemento.	46
Creando desde vacío.	46
Pertenencia de una clave.	47
Obtener todos los elementos.....	47
Longitud de un diccionario.....	47
Iterar sobre un diccionario.	48

Combinar diccionarios.....	48
Borrar elementos.	49
Borrado completo del diccionario.....	50
4.4 Conjuntos.	50
4.5 Ficheros.	50
4.5.1 Lectura de un fichero.	51
Lectura completa de un fichero.	51
Lectura línea a línea.	52
4.5.2 Escritura en un fichero.	52
4.5.3 Añadido a un fichero.....	53
4.5.4 Usandos contextos.	53
5. Estructuras de control de flujo.....	55
5.1 Aspectos previos.	55
5.1.1 Definición de bloques.....	55
5.1.2 Comentarios.	55
5.1.3 Ancho del código.....	55
5.2 Condicionales.	56
5.2.1 La sentencia if.....	56
5.2.2 Operadores de comparación.....	57
Variables y valores booleanos» en condiciones.....	58
Valor nulo.	59
5.2.3 Sentencia match-case.	59
Comparando valores.	59
Patrones avanzados.	60
5.3 Bucles.	62
5.3.1 La sentencia while.....	62
Romper un bucle while.	63
Comprobar la rotura.	63
Continuar un bucle.....	63
5.3.2 La sentencia for.	64
Romper un bucle for.	64
Secuencias de números.....	65
Uso del guión bajo.....	66
5.3.3 Bucles anidados.....	66
6. Modularidad.....	67
6.1 Funciones.	67

6.1.1 Definir una función.....	67
Invocar una función.....	68
Retornar un valor.	68
6.1.2 Parámetros y argumentos.....	68
Argumentos posicionales.	69
Argumentos nominales.	70
Argumentos posicionales y nominales.....	70
Parámetros por defecto.	70
Modificando parámetros mutables.	71
Empaquetar/Desempaquetar argumentos.....	72
Forzando modo de paso de argumentos.	73
Funciones como parámetros.....	75
6.1.3 Documentación.	75
Explicación de parámetros.	76
6.1.4 Tipos de funciones.	77
Funciones interiores.....	77
Clausuras.	77
Funciones anónimas «lambda».....	78
Decoradores.	79
Usando @ para decorar.	79
Funciones recursivas.	80
6.1.5 Espacios de nombres.....	81
Acceso a variables globales.	81
Creando variables locales.....	81
Forzando modificación global.	82
Contenido de los espacios de nombres.	82
6.2 Objetos y Clases.	82
6.2.1 Programación orientada a objetos.....	82
6.2.2 Creando objetos.	83
Añadiendo atributos.....	84
Añadiendo métodos.....	84
Inicialización.	85
6.2.3 Atributos.....	85
Acceso directo.	85
Propiedades.....	86
Valores calculados.	86

Ocultando atributos.	87
Atributos de clase.	87
6.2.4 Métodos.	88
Métodos de instancia.	88
Métodos de clase.	88
Métodos estáticos.	88
Métodos mágicos.	89
6.2.5 Herencia.	90
Heredar desde una clase base.	91
Sobreescribir un método.	91
Añadir un método.	92
Accediendo a la clase base.	92
Herencia múltiple.	93
Agregación y composición.	94
6.3 Módulos.	95
6.3.1 Importar un módulo.	95
Importar módulo completo.	96
Ruta de búsqueda de módulos.	96
Modificando la ruta de búsqueda.	96
Importar partes de un módulo.	97
Importar usando un alias.	97
6.3.2 Paquetes.	97
Importar desde un paquete.	98
6.3.3 Programa principal.	98
if __name__ == __main__	99
7. Excepciones.	100
7.1. Manejando errores.	100
Especificando excepciones.	100
Cubriendo más casos.	101
7.2 Excepciones propias.	102
Mensaje personalizado.	102
7.3 Aserciones.	103

1. Introducción a Python.

Python es un lenguaje de programación de alto nivel creado a finales de los 80 y principios de los 90 por Guido van Rossum (Holanda), que trabajaba por aquella época en el Centro para las Matemáticas y la Informática de los Países Bajos.

Sus instrucciones están muy cercanas al lenguaje natural en inglés y se hace hincapié en la legibilidad del código. Toma su nombre de los Monty Python, grupo humorista de los 60 que gustaban mucho a Guido. Python fue creado como sucesor del lenguaje ABC.

1.1 Características del lenguaje.

Python tiene las siguientes características:

- ✓ Python es un lenguaje de programación interpretado y multiplataforma cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.
- ✓ Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.
- ✓ Es un lenguaje de propósito general.

1.2 Ventajas.

Podemos destacar las siguientes ventajas de Python:

- ✓ Es libre y gratuito (OpenSource).
- ✓ Es fácil de leer (parecido a pseudocódigo).
- ✓ Tiene un aprendizaje relativamente fácil y rápido: claro, intuitivo....
- ✓ Es de alto nivel.
- ✓ Permite una alta Productividad: simple y rápido.
- ✓ Tiende a producir un buen código: orden, limpieza, elegancia, flexibilidad, ...
- ✓ Es multiplataforma (portable).
- ✓ Es ultiparadigma: programación imperativa, orientada a objetos, funcional, ...
- ✓ Es interactivo, modular, y dinámico.
- ✓ Ofrece librerías extensivas (pilas incluidas).
- ✓ Dispone de gran cantidad de librerías de terceros.
- ✓ Es extensible (C++, C, ...) y embebible.
- ✓ Tiene una gran comunidad y un amplio soporte.
- ✓ Es Interpretado.
- ✓ Tiene un tipado dinámico y es fuertemente tipado.
- ✓ Hay diferentes implementaciones: CPython, PyPy, Jython, IronPython, MicroPython,...

1.3 Desventajas

- ✓ Al ser interpretado, empeora su velocidad de ejecución.
- ✓ Tiene un consumo de memoria mayor que otros lenguajes.
- ✓ Se producen errores durante la ejecución.
- ✓ Existen dos versiones mayores que no son del todo compatibles (v2 vs v3).
- ✓ Tiene dificultades para el desarrollo para el móvil.
- ✓ Su documentación a veces es dispersa e incompleta.
- ✓ Puede tener varios módulos para la misma funcionalidad.
- ✓ Las librerías de terceros no siempre están del todo maduras.

1.4 Uso de Python.

Al ser un lenguaje de propósito general, podemos encontrar aplicaciones prácticamente en todos los campos científico-tecnológicos:

- ✓ Análisis de datos.
- ✓ Aplicaciones de escritorio.
- ✓ Bases de datos relacionales / NoSQL
- ✓ Buenas prácticas de programación / Patrones de diseño.
- ✓ Concurrencia.
- ✓ Criptomonedas / Blockchain.
- ✓ Desarrollo de aplicaciones multimedia.
- ✓ Desarrollo de juegos.
- ✓ Desarrollo en dispositivos embebidos.
- ✓ Desarrollo móvil.
- ✓ Desarrollo web.
- ✓ DevOps / Administración de sistemas / Scripts de automatización.
- ✓ Gráficos por ordenador.
- ✓ Inteligencia artificial.
- ✓ Internet de las cosas.
- ✓ Machine Learning.
- ✓ Programación de parsers / scrapers / crawlers.
- ✓ Programación de redes.
- ✓ Propósitos educativos.
- ✓ Prototipado de software.
- ✓ Seguridad.
- ✓ Tests automatizados.

De igual modo, son muchas las empresas, instituciones y organismos que utilizan Python en su día a día para mejorar sus sistemas de información. Entre ellas podemos destacar a Google, Spotify, DropBox, YouTube, Facebook, Paypal, Amazon, la NASA, la CIA, Netflix, IBM,...

Existen rankings y estudios de mercado que sitúan a Python como uno de los lenguajes más usados y, a la vez, más amados dentro del mundo del desarrollo de software.

1.5 Evolución de Python.

A continuación se muestra una tabla con la evolución del lenguaje por sus diferentes versiones:

Versión	Fecha de lanzamiento
Python 1.0	Enero 1994
Python 1.5	Diciembre 1997
Python 1.6	Septiembre 2000
Python 2.0	Octubre 2000
Python 2.1	Abril 2001
Python 2.2	Diciembre 2001
Python 2.3	Julio 2003
Python 2.4	Noviembre 2004
Python 2.5	Septiembre 2006
Python 2.6	Octubre 2008
Python 2.7	Julio 2010
Python 3.0	Diciembre 2008
Python 3.1	Junio 2009
Python 3.2	Febrero 2011

Python 3.3	Septiembre 2012
Python 3.4	Marzo 2014
Python 3.5	Septiembre 2015
Python 3.6	Diciembre 2016
Python 3.7	Junio 2018
Python 3.8	Octubre 2019
Python 3.9	Octubre 2020
Python 3.10	Octubre 2021

El cambio de Python 2 a Python 3 fue bastante traumático, ya que se perdió la compatibilidad en muchas de las estructuras del lenguaje. Sus principales desarrolladores vieron la necesidad de aplicar estas modificaciones en beneficio del rendimiento y expresividad del lenguaje de programación.

Este cambio implicaba que el código escrito en Python 2 no funcionaría (de manera inmediata) en Python 3. A partir de 2020 ha finalizado oficialmente el soporte de la versión 2.7 del lenguaje, por lo que se recomienda trabajar con la versión 3.

2. Instalación de Python y otros programas. Preparación del entorno de trabajo.

Cuando vamos a trabajar con Python debemos tener instalado, como mínimo, un intérprete del lenguaje (para otros lenguajes sería un compilador). El intérprete nos permitirá ejecutar nuestro código para obtener los resultados deseados. La idea del intérprete es lanzar “instrucciones sueltas” para probar determinados aspectos.

Pero normalmente queremos ir un poco más allá y poder escribir programas algo más largos, por lo que también necesitaremos un editor. Un editor es un programa que nos permite crear ficheros de código (en nuestro caso con extensión *.py), que luego son ejecutados por el intérprete.

Hay otra herramienta interesante dentro del entorno de desarrollo que sería el depurador. Es el módulo que nos permite ejecutar paso a paso nuestro código y visualizar qué está ocurriendo en cada momento. Se suele usar normalmente para encontrar fallos (bugs) en nuestros programas y poder solucionarlos (debug/fix).

Thonny es un programa muy interesante para empezar a aprender Python ya que engloba estas tres herramientas: intérprete, editor y depurador.

Si queremos otros programas más potentes podemos usar PyCharm, que es un IDE que incluye también el intérprete, el editor y el depurador, junto con muchas otras herramientas y utilidades.

2.1 Probar el intérprete.

El intérprete de Python (por lo general) se identifica claramente porque posee un prompt con tres ángulos hacia la derecha (>>>). Los programas e IDE que instalemos tendrán un panel para visualizar el intérprete, pero dicho intérprete es una herramienta autocontenida y la podemos ejecutar desde el símbolo del sistema o la terminal una vez que tenemos Python instalado:

```
$ python3.7
Python 3.7.4 (v3.7.4:e09359112e, Jul 8 2019, 14:54:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Para hacer una prueba inicial del intérprete vamos a ejecutar la famosa instrucción «Hello, World». Para ello escribimos lo siguiente en el intérprete y pulsamos la tecla ENTER:

```
>>> print('Hello, World')
Hello, World
```

Lo que hemos hecho es indicarle a Python que ejecute como entrada la instrucción “print(‘Hello, World’)”. La salida es ese texto, que se muestra en la siguiente línea (ya sin el prompt >>>).

2.2 Probar el editor.

Ahora vamos a realizar la misma operación, pero en vez de ejecutar la instrucción directamente en el intérprete, vamos a crear un fichero y guardarlo con la sentencia que nos interesa. Para ello escribimos instrucción “print(‘Hello, World’)” en el panel de edición del IDE y luego guardamos el archivo con el nombre “helloworld.py” (Los ficheros que contienen programas hechos en Python siempre deben tener la extensión .py).

Ahora ya podemos ejecutar nuestro fichero “helloworld.py”. Para ello pulsamos en el IDE el botón adecuado o la opción de menú que ejecute el programa. Veremos que, en el panel de Shell del IDE, nos aparece la salida esperada.

Lo que está pasando internamente es que el intérprete de Python está recibiendo como entrada el fichero que hemos creado, lo ejecuta, y devuelve la salida para que el IDE nos lo muestre en el panel correspondiente.

2.3 Probar el depurador.

Nos falta por probar el depurador o “debugger”. Aunque su funcionamiento va mucho más allá, de momento nos vamos a quedar con un par de detalles:

- ✓ Podemos ejecutar el programa paso a paso, es decir, instrucción a instrucción. El IDE tendrá alguna opción de menú para establecer este tipo de funcionamiento.
- ✓ Si existieran variables, podríamos ver el contenido de estas en el panel de depuración del IDE, y si esto lo unimos a la ejecución paso a paso, como va modificándose su contenido en cada momento, podremos ir viendo cómo evolucionan dichas variables.

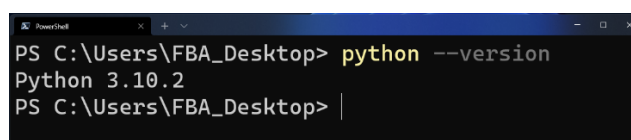
2.4 Instalación de Python.

La forma más habitual de instalar Python (junto con sus librerías) es descargarlo e instalarlo desde su página oficial, que es diferente para cada sistema operativo:

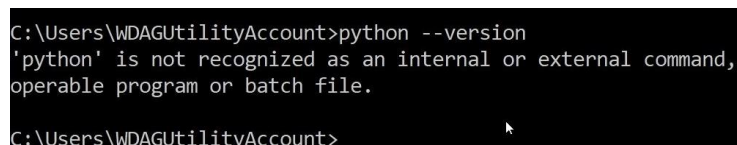
- ✓ Windows: <https://www.python.org/downloads/windows/>
- ✓ Linux: <https://www.python.org/downloads/source/>
- ✓ Mac: <https://www.python.org/downloads/mac-osx/>

Nosotros, en este caso, explicaremos sólo la instalación en Windows, pero es sencillo encontrar documentación de cómo instalarlo en los otros sistemas.

Lo primero que haremos será comprobar si tenemos Python instalado en nuestro sistema. Para ello, abriremos un terminal y ejecutaremos “python –versión”:



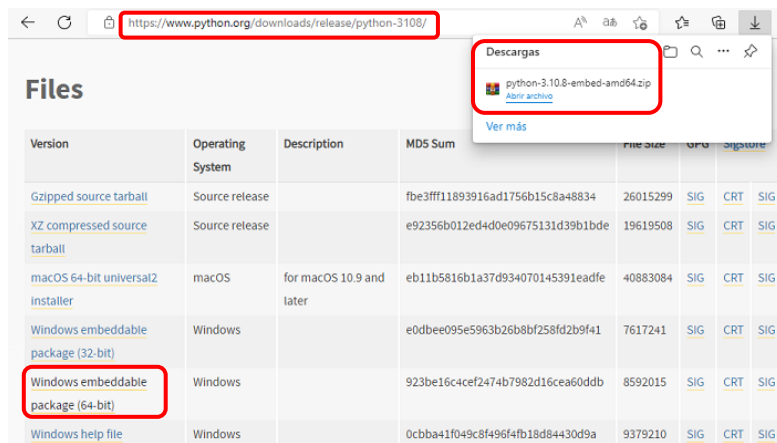
```
PS C:\Users\FBA_Desktop> python --version
Python 3.10.2
PS C:\Users\FBA_Desktop> |
```



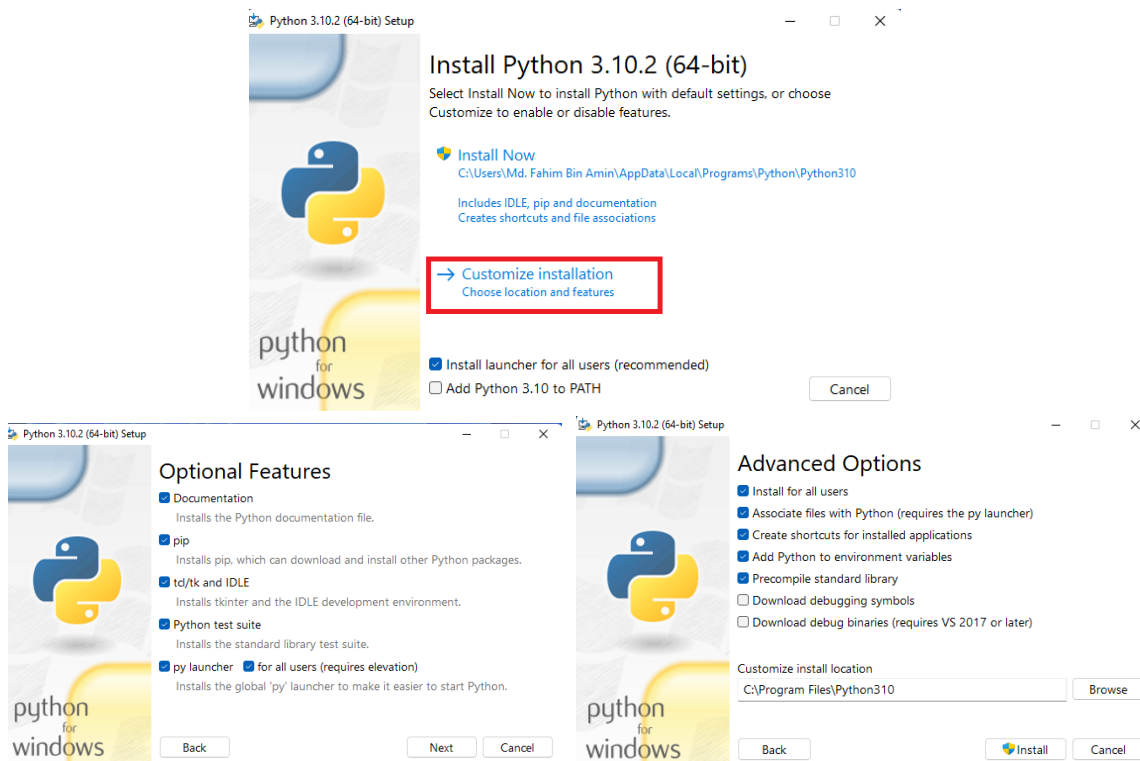
```
C:\Users\WDAGUtilityAccount>python --version
'python' is not recognized as an internal or external command,
operable program or batch file.
C:\Users\WDAGUtilityAccount>_
```

En la imagen de arriba aparece la versión instalada, por lo tanto, no tendremos que hacer nada, a no ser que queramos instalar otra versión más reciente. En la imagen de abajo nos indica que no se reconoce el comando, por lo que tendremos que instalar Python.

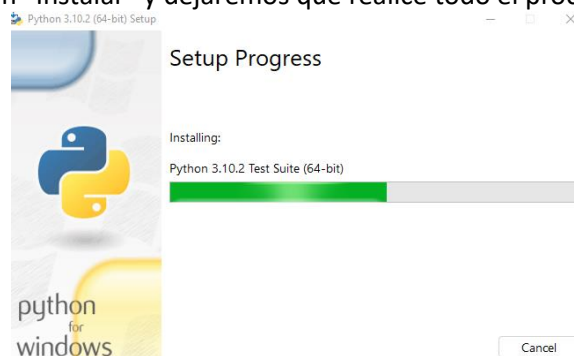
El primer paso para hacer la instalación es descargar el fichero de instalación desde la página de Python. Una vez allí, buscamos la versión que necesitamos (por ejemplo, la 3.10 de Windows). La siguiente imagen muestra cómo se hace la descarga para Windows de 64 bits:



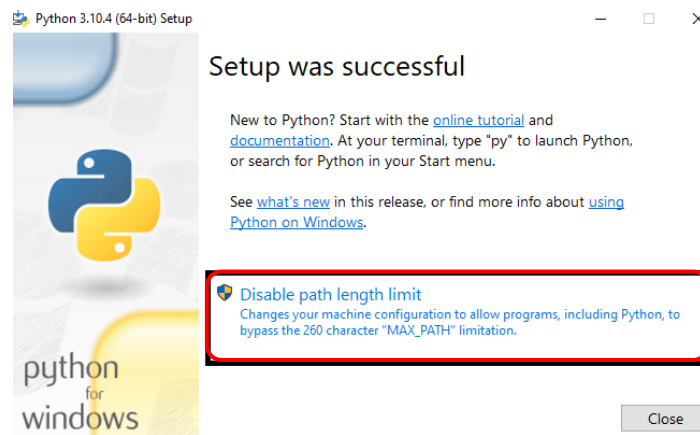
Después de descargar el archivo, obtendremos un archivo ejecutable en el que simplemente haremos doble clic para que se abra el asistente de instalación. En dicho asistente elegimos una instalación personalizada para poder elegir las características opcionales que queremos instalar (normalmente todas), las opciones avanzadas (normalmente todas) y el directorio de instalación (que normalmente se deja por defecto):



Después pulsaremos en “Instalar” y dejaremos que realice todo el proceso.



Es posible que, al finalizar la instalación, nos pregunte si queremos deshabilitar el límite de longitud de ruta eliminando la limitación de la variable MAX_PATH. Es conveniente hacer esto para permitir nombres largos de ruta. Simplemente tendremos que pulsar el botón que nos ofrece para conseguirlo:



Y con esto finalizaremos la instalación. Ya podremos volver a comprobar si está instalado Python en el terminal, y deberá aparecer la versión que acabamos de utilizar.

2.5 Gestión de paquetes.

La instalación limpia de Python ya ofrece de por sí muchos paquetes y módulos que vienen por defecto. Es lo que se llama la “librería estándar”. Pero una de las características más destacables de Python es su inmenso “ecosistema” de paquetes disponibles en el Python Package Index (PyPI).

Para gestionar los paquetes que tenemos en nuestro sistema se utiliza la herramienta pip, una utilidad que también se incluye en la instalación de Python. Con ella podremos instalar, desinstalar y actualizar paquetes, según nuestras necesidades. A continuación, se muestran las instrucciones que usaríamos para cada una de estas operaciones:

```
$ pip install pandas
$ pip uninstall pandas
$ pip install pandas --upgrade
```

2.6 Entornos virtuales.

Cuando trabajamos en distintos proyectos, no todos ellos requieren los mismos paquetes ni siquiera la misma versión de Python. La gestión de estas situaciones no es sencilla si únicamente instalamos paquetes y manejamos configuraciones a nivel global (*a nivel de máquina*). Es por ello que surge el concepto de **entornos virtuales**. Como su propio nombre indica, se trata de crear distintos entornos en función de las necesidades de cada proyecto, y esto nos permite establecer qué versión de Python usaremos y qué paquetes instalaremos.

La manera más sencilla de crear un entorno virtual es la siguiente:

```
$ cd myproject
$ python3 -m venv .env
$ source .env/bin/activate
```

El significado de cada línea es el siguiente:

- ✓ Línea 1: entrar en la carpeta de nuestro proyecto.
- ✓ Línea 2: crear una carpeta `.venv` con los ficheros necesarios que constituyen el entorno virtual.
- ✓ Línea 3: activar el entorno virtual. A partir de aquí todo lo que se instale quedará dentro del entorno virtual.

Otra manera de manejar los entornos virtuales es a través de “`virtualenv`”. Se trata de un paquete de Python que nos proporciona la funcionalidad de crear y gestionar entornos virtuales. Su instalación es sencilla a través del gestor de paquetes `pip`:

```
$ pip install virtualenv
```

También existen otras herramientas que nos permiten gestionar los entornos virtuales, incluso algunos IDE, como PyCharm, incluyen esta posibilidad.

3. Tipos de datos básicos en Python.

Igual que en el mundo real cada objeto pertenece a una categoría, en programación manejamos objetos que tienen asociado un tipo determinado. En esta sección se verán los tipos de datos básicos con los que podemos trabajar en Python.

A continuación, se muestra una tabla con estos tipos de datos, sin incluir aquellos que pueden proporcionarnos algunos paquetes externos:

Nombre	Tipo	Ejemplos
Booleano	bool	True, False
Entero	int	21, 34500, 34_500
Flotante	float	3.14, 1.5e3
Complejo	complex	2j, 3 + 5j
Cadena	str	'tfn', '''tenerife - islas canarias'''
Tupla	tuple	(1, 3, 5)
Lista	list	['Chrome', 'Firefox']
Conjunto	set	set([2, 4, 6])
Diccionario	dict	{'Chrome': 'v79' , 'Firefox': 'v71'}

3.1 Variables.

Las variables son fundamentales ya que permiten definir nombres para los valores que tenemos en memoria y que vamos a usar en nuestro programa.

En Python existe una serie de reglas para los nombres de variables:

- ✓ Sólo pueden contener los siguientes caracteres:
 - ❖ Letras minúsculas.
 - ❖ Letras mayúsculas.
 - ❖ Dígitos.
 - ❖ Guiones bajos (_).
- ✓ Deben empezar con una letra o un guión bajo, nunca con un dígito.
- ✓ No pueden ser una palabra reservada del lenguaje (keywords).

Podemos obtener un listado de las palabras reservadas del lenguaje de la siguiente forma:

```
>>> help('keywords')

Here is a list of the Python keywords. Enter any keyword to get more help.

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

Los nombres de variables son “case-sensitive”, es decir, distinguen entre mayúsculas y minúsculas. Por ejemplo, stuff y Stuff son nombres de variable diferentes.

Por lo general, se prefiere dar nombres en inglés a las variables que utilicemos, ya que así hacemos nuestro código más “internacional” y con la posibilidad de que otras personas puedan leerlo, entenderlo y, llegado el caso, modificarlo. Es sólo una recomendación, nada impide que se haga en castellano.

A continuación, se muestra una tabla con nombres de variable válidos e inválidos y, en el último caso, por qué no están permitidos:

Válido	Inválido	Razón
a	3	Empieza por un dígito
a3	3a	Empieza por un dígito
a_b_c__95	another-name	Contiene un caracter no permitido
_abc	with	Es una palabra reservada del lenguaje
_3a	3_a	Empieza por un dígito

Convenciones para nombres:

Mientras se sigan las reglas que hemos visto para nombrar variables no hay problema, pero sí existe una convención para la nomenclatura de las variables. Se utiliza el llamado “snake_case” en el que utilizamos caracteres en minúsculas (incluyendo dígitos si procede) junto con guiones bajos (cuando sean necesarios para su legibilidad). Por ejemplo, para nombrar una variable que almacene el número de canciones en nuestro ordenador, podríamos usar “num_songs”.

Esta convención, y muchas otras, están definidas en un documento denominado PEP 8. Se trata de una guía de estilo para escribir código en Python. Los PEPs son las propuestas que se hacen para la mejora del lenguaje.

Es muy importante elegir bien el nombre de las variables, ya que deben ser suficientemente explicativos sin que sean demasiado largos. Por tanto, habrá que llegar a un compromiso entre ser concisos y claros.

Otra regla que puede servirnos de ayuda puede ser:

- ✓ Usar nombres para variables (ejemplo article).
- ✓ Usar verbos para funciones (ejemplo get_article()).
- ✓ Usar adjetivos para booleanos (ejemplo available).

Constantes.

Un caso especial de variables son las constantes. Podríamos decir que es un tipo de variable pero que su valor no cambia a lo largo de nuestro programa. Por ejemplo, la velocidad de la luz. Sabemos que su valor es constante (300.000 km/s). En el caso de las constantes utilizamos mayúsculas (incluyendo guiones bajos si es necesario) para nombrarlas. Para el ejemplo de la velocidad de la luz la constante se podría llamar: LIGHT_SPEED.

3.2 Asignación de valores a variables.

En Python se usa el símbolo = para asignar un valor a una variable. No debemos confundir este símbolo con la igualdad de matemáticas que expresa equivalencia.

Algunos ejemplos de asignaciones a variables:

```
>>> total_population = 157503
>>> avg_temperature = 16.8
>>> city_name = 'San Cristóbal de La Laguna'
```


Algunos ejemplos de asignaciones a constantes:

```
>>> SOUND_SPEED = 343.2
>>> WATER_DENSITY = 997
>>> EARTH_NAME = 'La Tierra'
```

Python nos ofrece la posibilidad de hacer una asignación múltiple de la siguiente manera:

```
>>> tres = three = drei = 3
```

En este caso las tres variables utilizadas tomarán el valor 3.

Las asignaciones que hemos hecho hasta ahora han sido de un valor literal a una variable. Pero nada impide que podamos hacer asignaciones de una variable a otra variable:

```
>>> people = 157503
>>> total_population = people
>>> total_population
157503
```

Eso sí, la variable que utilicemos como valor de asignación debe existir previamente, ya que, si no es así, obtendremos un error informando de que no está definida.

Conocer el valor de una variable.

Para conocer el valor de una variable podemos utilizar dos estrategias:

Si estamos en una shell de Python, basta con que usemos el nombre de la variable:

```
>>> final_stock = 38934
>>> final_stock
38934
```

Si estamos escribiendo un programa desde un editor, podemos hacer uso de “print”:

```
final_stock = 38934
print(final_stock)
```

La función “print” sirve también cuando estamos en una Shell.

Conocer el tipo de una variable.

Para poder descubrir el tipo de un literal o una variable, Python nos ofrece la función type().

Veamos algunos ejemplos de su uso:

```
>>> type(9)
int
>>> type(1.2)
float
>>> height = 3718
>>> type(height)
int
>>> sound_speed = 343.2
>>> type(sound_speed)
float
>>> type(True)
bool
```

3.3 Números.

En esta sección veremos los tipos de datos numéricos que ofrece Python, centrándonos en booleanos, enteros y flotantes.

3.3.1 Booleanos.

El tipo de datos bool admite dos posibles valores:

- ✓ True: que se corresponde con verdadero (y también con 1 en su representación numérica).
- ✓ False: que se corresponde con falso (y también con 0 en su representación numérica).

Veamos un ejemplo de su uso:

```
>>> is_opened = True
>>> is_opened
True
>>> has_sugar = False
>>> has_sugar
False
```

La primera variable `is_opened` está representando el hecho de que algo esté abierto, y al tomar el valor `True` podemos concluir que sí. La segunda variable `has_sugar` nos indica si una bebida tiene azúcar; dado que toma el valor `False` inferimos que no lleva azúcar.

Atención: el tipo booleano toma los valores exactos `True` y `False` con la primera letra en mayúsculas. De no ser así, obtendríamos un error sintáctico.

3.3.2 Enteros.

Los números enteros no tienen decimales, pero sí pueden contener signo y estar expresados en alguna base distinta de la usual (base 10).

Veamos algunos ejemplos de números enteros:

```
>>> 8
8
>>> 0
0
>>> 08
File "<stdin>", line 1
08
^
SyntaxError: invalid token
>>> 99
99
>>> +99
99
>>> -99
-99
>>> 3000000
3000000
>>> 3_000_000
3000000
```

Debemos tener dos detalles en cuenta:

- ✓ No podemos comenzar un número entero por 0.
- ✓ Python permite dividir los números enteros con guiones bajos _ para clarificar su lectura/escritura. A efectos prácticos es como si esos guiones bajos no existieran.

Operaciones con enteros.

A continuación, se muestra una tabla con las distintas operaciones sobre enteros que podemos realizar en Python:

Operador	Descripción	Ejemplo	Resultado
+	Suma	3 + 9	12
-	Resta	6 - 2	4
*	Multiplicación	5 * 5	25
/	División flotante	9 / 2	4.5
//	División entera	9 // 2	4
%	Módulo	9 % 4	1
**	Exponenciación	2 ** 4	16

Veamos algunas pruebas de estos operadores:

```
>>> 2 + 8 + 4
14
>>> 4 ** 4
256
>>> 7 / 3
2.3333333333333335
>>> 7 // 3
2
>>> 6 / 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Se considera un buen estilo de programación dejar un espacio entre cada operador. Además, hay que tener en cuenta que podemos obtener errores dependiendo de la operación (más bien de los operandos) que estemos utilizando, como es el caso de la división por cero.

Asignación aumentada.

Consiste en mezclar la asignación con un operador. Python permite su uso. A continuación, se muestra un ejemplo de su uso:

```
>>> random_number = 15
>>> random_number += 5
>>> random_number
20
>>> random_number *= 3
>>> random_number
60
>>> random_number //= 4
>>> random_number
15
>>> random_number **= 1
>>> random_number
15
```

Valor absoluto

Python ofrece la función `abs()` para obtener el valor absoluto de un número:

```
>>> abs(-1)
1
>>> abs(1)
1
>>> abs(-3.14)
3.14
>>> abs(3.14)
3.14
```

3.3.3 Flotantes.

Los números en punto flotante³ tienen parte decimal. Veamos algunos ejemplos de flotantes en Python:

```
>>> 4.0
4.0
>>> 4.
4.0
>>> 04.0
4.0
>>> 04.
4.0
>>> 4.000_000
4.0
>>> 4e0 # 4.0 * (10 ** 0)
4.0
```

3.3.4 Conversión de tipos.

El hecho de que existan distintos tipos de datos en Python (y en el resto de los lenguajes de programación) es una ventaja a la hora de representar la información del mundo real de la mejor manera posible. Pero también se hace necesario buscar mecanismos para convertir unos tipos de datos en otros.

Conversión implícita.

Cuando mezclamos enteros, booleanos y flotantes, Python realiza automáticamente una conversión implícita (o promoción) de los valores al tipo de “mayor rango”. Veamos algunos ejemplos de esto:

```
>>> True + 25
26
>>> 7 * False
0
>>> True + False
1
>>> 21.8 + True
22.8
>>> 10 + 11.3
21.3
```

Tipo 1	Tipo 2	Resultado
bool	int	int
bool	float	float
int	float	float

Podemos resumir la conversión implícita según lo expuesto en la tabla.

Conversión explícita

Aunque más adelante veremos el concepto de función, desde ahora podemos decir que existen una serie de funciones para realizar conversiones explícitas de un tipo a otro:

- ✓ `bool()`: convierte el tipo a booleano.
- ✓ `int()`: convierte el tipo a entero.
- ✓ `float()`: convierte el tipo a flotante.

Veamos algunos ejemplos de estas funciones:

```
>>> bool(1)
True
>>> bool(0)
False
>>> int(True)
1
>>> int(False)
0
>>> float(1)
1.0
>>> float(0)
0.0
>>> float(True)
1.0
>>> float(False)
0.0
>>> int(3.1)
3
>>> int(3.9)
3
```

3.3.5 Bases numéricas.

Los valores numéricos con los que estamos acostumbrados a trabajar están en base 10 (o decimal). Pero también es posible representar números en otras bases. Python nos ofrece una serie de prefijos y funciones para este cometido.

Base binaria.

Cuenta con 2 símbolos para representar los valores: 0 y 1. Para expresar cantidades en binario comenzaremos por “0b”:

```
>>> 0b1001
9
>>> 0b1100
12
```

Existe una función llamada `bin()`, que transforma de base decimal a binaria:

```
>>> bin(9)
'0b1001'
>>> bin(12)
'0b1100'
```

Base octal.

Cuenta con 8 símbolos para representar los valores: 0, 1, 2, 3, 4, 5, 6 y 7. Para expresar cantidades en octal comenzaremos por "0o":

```
>>> 0o6243
3235
>>> 0o1257
687
```

Existe una función llamada `oct()`, que transforma de base decimal a octal:

```
>>> oct(3235)
'0o6243'
>>> oct(687)
'0o1257'
```

Base hexadecimal.

Cuenta con 16 símbolos para representar los valores: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F. Para expresar cantidades en hexadecimal comenzaremos por "0x":

```
>>> 0x7F2A
32554
>>> 0x48FF
18687
```

Existe una función llamada `hex()`, que transforma de base decimal a hexadecimal:

```
>>> hex(32554)
'0x7f2a'
>>> hex(18687)
'0x48ff'
```

Nota: Las letras para la representación hexadecimal no distingue entre mayúsculas y minúsculas.

3.4 Cadenas de texto.

3.4.1 Introducción a las cadenas de texto.

Las cadenas de texto son secuencias de caracteres. También se les conoce como «strings» y nos permiten almacenar información textual de forma muy cómoda.

Es importante destacar que Python 3 almacena los caracteres codificados en el estándar Unicode, lo que es una gran ventaja con respecto a versiones antiguas del lenguaje. Además, permite representar una cantidad ingente de símbolos incluyendo los famosos emojis.

Para escribir una cadena de texto en Python basta con rodear los caracteres con comillas simples o con comillas dobles:

```
>>> 'Mi primera cadena en Python'
'Mi primera cadena en Python'
```

Para incluir comillas dobles dentro de la cadena de texto, simplemente las escribimos. También se puede realizar al contrario (comillas simples dentro de dobles):

```
>>> 'Los llamados "strings" son secuencias de caracteres'
'Los llamados "strings" son secuencias de caracteres'
```

Si necesitamos incluir comillas simples dentro de otras comillas simples usaremos esta técnica:

```
>>> 'Los llamados \'strings\' son secuencias de caracteres'
"Los llamados 'strings' son secuencias de caracteres"
```

Hay una forma alternativa de crear cadenas de texto utilizando comillas triples. Su uso está pensado principalmente para cadenas multilínea:

```
>>> texto = '''Esta es la primera línea
... y esta es la segunda línea,
... y esta la tercera,
... y con esta terminamos'''
```

Importante: los tres puntos (...) que aparecen a la izquierda de las líneas no están incluidos en la cadena de texto. Es el símbolo que ofrece el intérprete de Python cuando saltamos de línea.

La cadena vacía es aquella que no contiene ningún carácter. Aunque, a priori, no lo pueda parecer, es un recurso importante en cualquier código. Su representación en Python es "".

Podemos crear «strings» a partir de otros tipos de datos usando la función str():

```
>>> str(True)
'True'
>>> str(10)
'10'
>>> str(21.7)
'21.7'
```

Secuencias de escape.

Python permite escapar el significado de algunos caracteres para conseguir otros resultados. Si escribimos una barra invertida \ antes del carácter en cuestión, le otorgamos un significado especial.

Quizás la secuencia de escape más conocida es \n que representa un salto de línea, pero existen muchas otras:

```
# Salto de línea
>>> msg = 'Primera línea\nSegunda línea\nTercera línea'
>>> print(msg)
Primera línea
Segunda línea
Tercera línea
# Tabulador
>>> msg = 'Valor = \t40'
>>> print(msg)
Valor =      40
# Comilla simple
>>> msg = 'Necesitamos \'escapar\' la comilla simple'
>>> print(msg)
Necesitamos 'escapar' la comilla simple
# Barra invertida
>>> msg = 'Capítulo \\ Sección \\ Encabezado'
>>> print(msg)
Capítulo \ Sección \ Encabezado
```

Si en algún momento queremos que los caracteres especiales pierdan ese significado y queremos usar el texto “en bruto”, antepondremos una “r” a la cadena de texto:

```
>>> text = 'abc\ndef'
>>> print(text)
abc
def
>>> text = r'abc\ndef'
>>> print(text)
abc\ndef
```

Uso avanzado de print().

Hemos estado utilizando la función print() de forma sencilla, pero admite algunos parámetros interesantes:

```
1 >>> msg1 = '¿Sabes por qué estoy acá?'
2 >>> msg2 = 'Porque me apasiona'
3
4 >>> print(msg1, msg2)
5 ¿Sabes por qué estoy acá? Porque me apasiona
6
7 >>> print(msg1, msg2, sep='|')
8 ¿Sabes por qué estoy acá?| Porque me apasiona
9
10 >>> print(msg2, end='!!')
11 Porque me apasiona!!
```

Aclaraciones:

- ✓ Línea 4: podemos imprimir todas las variables que queramos separándolas por comas.
- ✓ Línea 7: el separador por defecto entre las variables es un espacio, pero podemos cambiar el carácter que se utiliza como separador entre cadenas.
- ✓ Línea 10: el carácter de final de texto es un salto de línea, pero podemos cambiar el carácter que se utiliza como final de texto.

Leer datos desde teclado.

Los programas se hacen para tener interacción con el usuario. Una de las formas de interacción es solicitar la entrada de datos por teclado. Como muchos otros lenguajes de programación, Python también nos ofrece la posibilidad de leer la información introducida por teclado. Para ello se utiliza la función input():

```
>>> name = input('Introduzca su nombre: ')
Introduzca su nombre: Sergio
>>> name
'Sergio'
>>> type(name)
str
>>> age = input('Introduzca su edad: ')
Introduzca su edad: 41
>>> age
'41'
>>> type(age)
str
```

Nota: La función input() siempre nos devuelve un objeto de tipo cadena de texto o str. Debemos tenerlo muy en cuenta a la hora de trabajar con números, ya que debemos realizar una conversión explícita.

3.4.2 Operaciones con cadenas de texto.

Combinar cadenas.

Podemos combinar dos o más cadenas de texto utilizando el operador +:

```
>>> proverb1 = 'Cuando el río suena'  
>>> proverb2 = 'agua lleva'  
>>> proverb1 + proverb2  
'Cuando el río suenaagua lleva'  
>>> proverb1 + ' , ' + proverb2 # incluimos una coma  
'Cuando el río suena, agua lleva'
```

Repetir cadenas.

Podemos repetir dos o más cadenas de texto utilizando el operador *:

```
>>> saludo = 'hola'  
>>> saludo * 4  
'holaholaholahola'
```

Obtener un carácter.

Los «strings» están indexados y cada carácter tiene su propia posición. Para obtener un único carácter dentro de una cadena de texto es necesario especificar su índice dentro de corchetes:

```
>>> sentence = 'Hola, Mundo'  
>>> sentence[0]  
'H'  
>>> sentence[-1]  
'o'  
>>> sentence[4]  
,  
>>> sentence[-5]  
'M'
```

Tengamos en cuenta que el indexado de una cadena de texto siempre empieza en 0 y termina en una unidad menos de la longitud de la cadena. Además, fijémonos que existen tanto índices positivos como índices negativos para acceder a cada carácter de la cadena de texto. A priori puede parecer redundante, pero es muy útil en determinados casos.

En caso de que intentemos acceder a un índice que no existe, obtendremos un error por fuera de rango.

Las cadenas de texto son tipos de datos inmutables. Es por ello que no podemos modificar un carácter directamente:

```
>>> song = 'Hey Jude'  
>>> song[4] = 'D'  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: 'str' object does not support item assignment
```

Más adelante se verá como se puede resolver este problema.

Trocear una cadena.

Es posible extraer «trozos» («rebanadas») de una cadena de texto. Tenemos varias aproximaciones para ello:

- ✓ [:] Extrae la secuencia entera desde el comienzo hasta el final. Es una especie de copia de toda la cadena de texto.
- ✓ [start:] Extrae desde start hasta el final de la cadena.
- ✓ [:end] Extrae desde el comienzo de la cadena hasta end menos 1.
- ✓ [start:end] Extrae desde start hasta end menos 1.
- ✓ [start:end:step] Extrae desde start hasta end menos 1 haciendo saltos de tamaño step.

Veamos la aplicación de cada uno de estos accesos a través de un ejemplo:

```
>>> proverb = 'Agua pasada no mueve molino'
>>> proverb[:]
'Agua pasada no mueve molino'
>>> proverb[12:]
'no mueve molino'
>>> proverb[:11]
'Agua pasada'
>>> proverb[5:11]
'pasada'
>>> proverb[5:11:2]
'psd'
```

Importante: El troceado siempre llega a una unidad menos del índice final que hayamos especificado. Sin embargo, el comienzo sí coincide con el que hemos puesto.

Longitud de una cadena.

Para obtener la longitud de una cadena podemos hacer uso de len(), una función común a prácticamente todos los tipos y estructuras de datos en Python:

```
>>> proverb = 'Lo cortés no quita lo valiente'
>>> len(proverb)
27
>>> empty = ''
>>> len(empty)
0
```

Pertenencia de un elemento.

Si queremos comprobar que una determinada subcadena se encuentra en una cadena de texto utilizamos el operador in para ello. Se trata de una expresión que tiene como resultado un valor «booleano» verdadero o falso:

```
>>> proverb = 'Más vale malo conocido que bueno por conocer'
>>> 'malo' in proverb
True
>>> 'bueno' in proverb
True
>>> 'regular' in proverb
False
```

Dividir una cadena.

Una tarea muy común al trabajar con cadenas de texto es dividir las por algún tipo de separador. En este sentido, Python nos ofrece la función split(), que debemos usar anteponiendo el «string» que queramos dividir (es una notación de tipo objeto.metodo):

```
>>> proverb = 'No hay mal que por bien no venga'
>>> proverb.split()
['No', 'hay', 'mal', 'que', 'por', 'bien', 'no', 'venga']
```

```
>>> tools = 'Martillo,Sierra,Destornillador'
>>> tools.split(',')
['Martillo', 'Sierra', 'Destornillador']
```

Si no se especifica un separador, `split()` usa por defecto cualquier secuencia de espacios en blanco, tabuladores y saltos de línea para separar los trozos.

Aunque aún no lo hemos visto, lo que devuelve `split()` es una lista (otro tipo de datos en Python) donde cada elemento es una parte de la cadena de texto original.

Limpiar cadenas.

Cuando leemos datos del usuario o de cualquier fuente externa de información, es bastante probable que se incluyan en esas cadenas de texto ciertos caracteres de relleno al comienzo y al final. Python nos ofrece la posibilidad de eliminar estos caracteres u otros que no nos interesen.

La función `strip()` se utiliza para eliminar caracteres del principio y del final de un «string». También existen variantes de esta función para aplicarla únicamente al comienzo (`lstrip()`) o únicamente al final (`rstrip()`) de la cadena de texto.

Supongamos que debemos procesar un fichero con números de serie de un determinado artículo. Cada línea contiene el valor que nos interesa, pero se han «colado» ciertos caracteres de relleno que debemos limpiar:

```
>>> serial_number = '\n\t\n 48374983274832 \n\n\t\t\n'
>>> serial_number.strip()
'48374983274832'
```

Si no se especifican los caracteres a eliminar, `strip()` usa por defecto cualquier combinación de espacios en blanco, saltos de línea (`\n`) y tabuladores (`\t`).

A continuación, veamos unos ejemplos de «limpieza» por la izquierda (comienzo) y por la derecha (final):

```
>>> serial_number.lstrip()
'48374983274832 \n\n\t\t\n'
>>> serial_number.rstrip()
'\n\t\n 48374983274832'
```

Como habíamos comentado, también existe la posibilidad de especificar los caracteres que queremos borrar:

```
>>> serial_number.strip('\n')
'\t\n 48374983274832 \n\n\t\t'
```

Importante: La función `strip()` no modifica la cadena que estamos usando (algo obvio porque los «strings» son inmutables) sino que devuelve una nueva cadena de texto con las modificaciones pertinentes.

Realizar búsquedas.

Python nos ofrece distintas alternativas para realizar búsquedas en cadenas de texto.

Vamos a partir de una variable que contiene un trozo de la canción Mediterráneo de Joan Manuel Serrat para realizar los ejemplos:

```
>>> lyrics = '''Quizás porque mi niñez
```

```
... Sigue jugando en tu playa
... Y escondido tras las cañas
... Duerme mi primer amor
... Llevo tu luz y tu olor
... Por dondequiera que vaya'''
```

Comprobar si una cadena de texto empieza o termina por alguna subcadena:

```
>>> lyrics.startswith('Quizás')
True
>>> lyrics.endswith('Final')
False
```

Encontrar la primera ocurrencia de alguna subcadena:

```
>>> lyrics.find('amor')
93
>>> lyrics.index('amor') # ¿Tienen el mismo comportamiento?
93
```

Tanto find() como index() devuelven el índice de la primera ocurrencia de la subcadena que estemos buscando, pero se diferencian en su comportamiento cuando la subcadena buscada no existe:

```
>>> lyrics.find('universo')
-1
>>> lyrics.index('universo')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Contabilizar el número de veces que aparece una subcadena:

```
>>> lyrics.count('mi')
2
>>> lyrics.count('él')
0
```

Reemplazar elementos.

Podemos usar la función replace() indicando la subcadena a reemplazar, la subcadena de reemplazo y cuántas instancias se deben reemplazar. Si no se especifica este último argumento, la sustitución se hará en todas las instancias encontradas:

```
>>> proverb = 'Quien mal anda mal acaba'
>>> proverb.replace('mal', 'bien')
'Quien bien anda bien acaba'
>>> proverb.replace('mal', 'bien', 1) # sólo 1 reemplazo
'Quien bien anda mal acaba'
```

Mayúsculas y minúsculas.

Python nos permite realizar variaciones en los caracteres de una cadena de texto para pasarlos a mayúsculas y/o minúsculas. Veamos las distintas opciones disponibles:

```
>>> proverb = 'quien a buen árbol se arrima Buena Sombra le cobija'
>>> proverb
'quien a buen árbol se arrima Buena Sombra le cobija'
```

```
>>> proverb.capitalize()
'Quien a buen árbol se arrima buena sombra le cobija'
>>> proverb.title()
'Quien A Buen Árbol Se Arrima Buena Sombra Le Cobija'
>>> proverb.upper()
'QUIEN A BUEN ÁRBOL SE ARRIMA BUENA SOMBRA LE COBIJA'
>>> proverb.lower()
'quien a buen árbol se arrima buena sombra le cobija'
>>> proverb.swapcase()
'QUIEN A BUEN ÁRBOL SE ARRIMA bUENA sOMBRA LE COBIJA'
```

Identificando caracteres.

Hay veces que recibimos información textual de distintas fuentes de las que necesitamos identificar qué tipo de caracteres contienen. Para ello Python nos ofrece un grupo de funciones:

```
>>> 'R2D2'.isalnum()    # Detectar si todos los caracteres son letras o números
True
>>> 'C3-PO'.isalnum()
False
>>> '314'.isnumeric()   # Detectar si todos los caracteres son números
True
>>> '3.14'.isnumeric()
False
>>> 'abc'.isalpha()      # Detectar si todos los caracteres son letras
True
>>> 'a-b-c'.isalpha()
False
>>> 'BIG'.isupper()      # Detectar mayúsculas/minúsculas
True
>>> 'small'.islower()
True
>>> 'First Heading'.istitle()
True
```

Interpolación de cadenas.

Interpolación (en este contexto) significa sustituir una variable por su valor dentro de una cadena de texto. Python soporta varios estilos para este cometido:

Nombre	Símbolo	Soportado
Estilo antiguo	%	>= Python2
Estilo «nuevo»	.format	>= Python2.6
«f-strings»	f''	>= Python3.6

Aunque aún podemos encontrar código con el estilo antiguo y el estilo nuevo en el formateo de cadenas, vamos a centrarnos en el análisis de los «f-strings» que son las que se utilizan en la actualidad.

Los f-strings son la forma más potente (y en muchas ocasiones más eficiente) de formar cadenas de texto incluyendo valores de otras variables. Para indicar en Python que una cadena es de tipo «f-string» basta con precederla de una f e incluir las variables o expresiones a interpolar entre llaves {...}.

Supongamos que disponemos de los datos de una persona y queremos formar una frase de bienvenida con ellos:

```
>>> name = 'Elon Musk'
>>> age = 49
>>> fortune = 43_300
>>> f'Me llamo {name}, tengo {age} años y una fortuna de {fortune} millones'
'Me llamo Elon Musk, tengo 49 años y una fortuna de 43300 millones'
```

Podría surgir la duda de cómo incluir llaves dentro de la cadena de texto, teniendo en cuenta que las llaves son símbolos especiales para la interpolación de variables. La respuesta es duplicando las llaves:

```
>>> x = 10
>>> f'The variable is {{ x = {x} }}'
'The variable is { x = 10 }'
```

Formateando cadenas.

Los «f-strings» proporcionan una gran variedad de opciones de formateado: ancho del texto, número de decimales, tamaño de la cifra, alineación, etc. Veamos algunos ejemplos:

```
>>> mount_height = 3718
>>> f'{mount_height:10d}'      #formato decimal con 10 caracteres rellenando con espacios
'  3718'
>>> f'{mount_height:010d}'     #formato decimal con 10 caracteres rellenando con ceros
'0000003718'
```

```
>>> value = 100
>>> f'{value}'                #si no se indica formato, se visualiza en decimal
'100'
>>> f'{value:d}'              #indicando expresamente el formato decimal
'100'
>>> f'{value:b}'              #indicando formato binario
'1100100'
>>> f'{value:o}'              #indicando formato octal
'144'
>>> f'{value:x}'              #indicando formato hexadecimal
'64'
```

```
>>> pi = 3.14159265
>>> f'{pi:f}'                 # 6 decimales por defecto (se rellenan con ceros si procede)
'3.141593'
>>> f'{pi:.3f}'               #no se indica parte entera, pero si 3 posiciones decimales
'3.142'
>>> f'{pi:12f}'               #12 caracteres en total (no se indica parte decimal)
'  3.141593'
>>> f'{pi:7.2f}'              #7 caracteres para la parte entera y 2 para la parte decimal
'  3.14'
>>> f'{pi:.010f}'             #sin configurar parte entera y 10 caracteres en parte decimal con relleno de ceros
'3.1415926500'
```

```
>>> text1 = 'how'
>>> text2 = 'are'
>>> text3 = 'you'
```

```
>>>#text1 alineado a la izquierda ocupando 7 caracteres
>>>#text2 alineado al centro ocupando 11 caracteres
>>>#text3 alineado a la derecha ocupando 7 caracteres
>>> f'{text1:<7s}|{text2:^11s}|{text3:>7s}'
'how | are | you'
>>>#mismo ejemplo pero con caracteres de relleno
>>> f'{text1:<-<7s}|{text2:~^11s}|{text3:->7s}'
'how----|.....are.....|----you'
```

Modo debug.

A partir de Python 3.8, los «f-strings» permiten imprimir el nombre de la variable junto con su valor como un atajo para depurar nuestro código. Para ello sólo tenemos que incluir un símbolo “=” después del nombre de la variable:

```
>>> serie = 'The Simpsons'
>>> imdb_rating = 8.7
>>> num_seasons = 30
>>> f'{serie=}'
"serie=The Simpsons"
>>> f'{imdb_rating=}'
'imdb_rating=8.7'
>>> f'{serie[4:]=}' # incluso podemos añadir expresiones
"serie[4:]=Simpsons"
>>> f'{imdb_rating / num_seasons=}' #y realizar operaciones matemáticas
'imdb_rating / num_seasons=0.29'
```

Casos de uso - dir().

Python ofrece una función interna llamada “dir()”, que muestra todas las funciones (casos de uso) que podemos aplicar a una determinada variable de tipo cadena:

```
>>> text = 'Hola'
>>> dir(text)
>>>#aquí aparecería la lista de todas las funciones aplicables
```

La función dir() no sólo es aplicable a las cadenas, sino a todos los tipos de variables, incluso a los literales y a los tipos de datos explícitos (clases):

```
>>> dir(10)
>>> dir(True)
>>> dir(float)
```

4. Colecciones en Python. Estructuras de datos.

Python puede manejar tipos de datos más complejos que forman colecciones o estructuras de datos basados en los tipos de datos básicos.

Entre estas colecciones encontraremos listas, tuplas, diccionarios y conjuntos (incluso ficheros).

4.1 Listas.

Las listas permiten almacenar objetos mediante un orden definido y con posibilidad de duplicados. Las listas son estructuras de datos mutables, lo que significa que podemos añadir, eliminar o modificar sus elementos.

Una lista está compuesta por cero o más elementos. En Python debemos escribir estos elementos separados por comas y dentro de corchetes. Una lista puede contener tipos de datos heterogéneos, lo que la hace una estructura de datos muy versátil.

Veamos algunos ejemplos de listas:

```
>>> empty_list = []
>>> languages = ['Python', 'Ruby', 'Javascript']
>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]
>>> data = ['Tenerife', {'cielo': 'limpio', 'temp': 24}, 3718, (28.2933947, -16.5226597)]
```

Para convertir otros tipos de datos en una lista podemos usar la función `list()`:

```
>>> # conversión desde una cadena de texto
>>> list('Python')
['P', 'y', 't', 'h', 'o', 'n']
```

Si nos fijamos en lo que ha pasado, al convertir la cadena de texto Python se ha creado una lista con 6 elementos, donde cada uno de ellos representa un carácter de la cadena.

Podemos extender este comportamiento a cualquier otro tipo de datos que permita ser iterado (iterables).

4.1.1 Operaciones con listas.

Obtener un elemento

Igual que en el caso de las cadenas de texto, podemos obtener un elemento de una lista a través del índice (lugar) que ocupa:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> shopping[0]
'Agua'
>>> shopping[2]
'Aceite'
>>> shopping[-2] # acceso con índice negativo
'Huevos'
```

El índice que usemos para acceder a los elementos de una lista tiene que estar comprendido entre los límites de la misma. Sino, obtendremos un error (excepción):

```
>>> shopping[3]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```


Trocear una lista.

El troceado de listas funciona de manera totalmente análoga al troceado de cadenas:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping[0:3]
['Agua', 'Huevos', 'Aceite']
>>> shopping[:3]
['Agua', 'Huevos', 'Aceite']
>>> shopping[-1:-4:-1]
['Limón', 'Sal', 'Aceite']
```

En el troceado de listas, a diferencia de lo que ocurre al obtener elementos, no debemos preocuparnos por acceder a índices inválidos (fuera de rango) ya que Python los restringirá a los límites de la lista.

Importante: Ninguna de las operaciones anteriores modifican la lista original, simplemente devuelven una lista nueva.

Invertir una lista.

Python nos ofrece, al menos, tres mecanismos para invertir los elementos de una lista:

Conservando la lista original, mediante troceado de listas con step negativo:

```
>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

Conservando la lista original, mediante la función `reversed()`:

```
>>> list(reversed(shopping))
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

Modificando la lista original, utilizando la función `reverse()` (nótese que es sin «d» al final):

```
>>> shopping.reverse()
>>> shopping
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

Añadir al final de la lista.

Una de las operaciones más utilizadas en listas es añadir elementos al final de las mismas. Para ello, Python nos ofrece la función `append()`. Se trata de un método destructivo que modifica la lista original:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> shopping.append('Atún')
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Atún']
```

Una forma muy habitual de trabajar con listas es empezar con una vacía e ir añadiendo elementos poco a poco. Se podría hablar de un patrón de creación. Supongamos un ejemplo en el que queremos construir una lista con los números pares del 1 al 20:

```
>>> even_numbers = []
>>> for i in range(20):
...     if i % 2 == 0:
...         even_numbers.append(i)
...
>>> even_numbers
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Añadir en cualquier posición de una lista.

Python ofrece una función `insert()`, que vendría a ser una generalización de `append()`, para incorporar elementos en cualquier posición. Simplemente debemos especificar el índice de inserción y el elemento en cuestión. También se trata de una función destructiva:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> shopping.insert(1, 'Jamón')
>>> shopping
['Agua', 'Jamón', 'Huevos', 'Aceite']
>>> shopping.insert(3, 'Queso')
>>> shopping
['Agua', 'Jamón', 'Huevos', 'Queso', 'Aceite']
```

No obtendremos un error si especificamos índices fuera de los límites de la lista. Estos se ajustarán al principio o al final en función del valor que indiquemos.

Repetir elementos.

Al igual que con las cadenas de texto, el operador `*` nos permite repetir los elementos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> shopping * 3
['Agua',
'Huevos',
'Aceite',
'Agua',
'Huevos',
'Aceite',
'Agua',
'Huevos',
'Aceite',]
```

Combinar listas.

Python nos ofrece dos formas para combinar listas:

Conservando la lista original, mediante el operador `+` o `+=`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']
>>> shopping + fruitshop
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

Modificando la lista original, mediante la función `extend()`:

```
>>> shopping.extend(fruitshop)
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

Hay que tener en cuenta que `extend()` funciona adecuadamente sólo si pasamos una lista como argumento. En otro caso, los resultados no serán los esperados.

Modificar una lista.

Del mismo modo que se accede a un elemento utilizando su índice, también podemos modificarlo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> shopping[0] = 'Jugo'
```

```
>>> shopping
['Jugo', 'Huevos', 'Aceite']
```

No sólo es posible modificar un elemento de cada vez, sino que podemos asignar valores a trozos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping[1:4]
['Huevos', 'Aceite', 'Sal']
>>> shopping[1:4] = ['Atún', 'Pasta']
>>> shopping
['Agua', 'Atún', 'Pasta', 'Limón']
```

Nota: La lista que asignamos no necesariamente debe tener la misma longitud que el trozo que sustituimos.

Borrar elementos.

Python nos ofrece, al menos, cuatro formas para borrar elementos en una lista:

Por su índice, mediante la función `del()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> del(shopping[3])
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

Por su valor, mediante la función `remove()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping.remove('Sal')
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

Advertencia: Si existen valores duplicados, la función `remove()` sólo borrará la primera ocurrencia.

Por su índice (con extracción): las dos funciones anteriores `del()` y `remove()` efectivamente borran el elemento indicado de la lista, pero no «devuelven» nada. Sin embargo, Python nos ofrece la función `pop()` que, además de borrar, nos «recupera» el elemento (algo así como una extracción). Lo podemos ver como una combinación de acceso + borrado:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping.pop()
'Limón'
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal']
>>> shopping.pop(2)
'Aceite'
>>> shopping
['Agua', 'Huevos', 'Sal']
```

Nota: Si usamos la función `pop()` sin pasarle ningún argumento, por defecto usará el índice -1, es decir, el último elemento de la lista. Pero también podemos indicarle el índice del elemento a extraer.

Por su rango, mediante troceado de listas:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping[1:4] = []
>>> shopping
['Agua', 'Limón']
```

Borrado completo de la lista.

Python nos ofrece, al menos, dos formas para borrar una lista por completo:

Utilizando la función `clear()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping.clear() # Borrado in-situ
>>> shopping
[]
```

«Reinicializando» la lista a vacío con `[]`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping = [] # Nueva zona de memoria
>>> shopping
[]
```

Encontrar un elemento.

Si queremos descubrir el índice que corresponde a un determinado valor dentro la lista podemos usar la función `index()` para ello:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping.index('Huevos')
1
```

Hay que tener en cuenta que si el elemento que buscamos no está en la lista, obtendremos un error:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping.index('Pollo')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 'Pollo' is not in list
```

Nota: Si buscamos un valor que existe más de una vez en una lista, la función `index()` sólo nos devolverá el índice de la primera ocurrencia.

Pertenencia de un elemento.

Si queremos comprobar la existencia de un determinado elemento en una lista, podríamos buscar su índice, pero la forma pitónica de hacerlo es utilizar el operador `in`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> 'Aceite' in shopping
True
>>> 'Pollo' in shopping
False
```

Nota: El operador `in` siempre devuelve un valor booleano, es decir, verdadero o falso.

Número de ocurrencias.

Para contar cuántas veces aparece un determinado valor dentro de una lista podemos usar la función `count()`:

```
>>> sheldon_greeting = ['Penny', 'Penny', 'Penny']
>>> sheldon_greeting.count('Howard')
0
>>> sheldon_greeting.count('Penny')
3
```

Convertir lista a cadena de texto.

Dada una lista, podemos convertirla a una cadena de texto, uniendo todos sus elementos mediante algún separador. Para ello hacemos uso de la función `join()` de la siguiente manera:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> ','.join(shopping)
'Agua,Huevos,Aceite,Sal,Limón'
>>> ' '.join(shopping)
'Agua Huevos Aceite Sal Limón'
>>> '|'.join(shopping)
'Agua|Huevos|Aceite|Sal|Limón'
```

Hay que tener en cuenta que `join()` sólo funciona si todos sus elementos son cadenas de texto. Además, debemos saber que la función `join()` es realmente la opuesta a la de `split()` para dividir una cadena.

Ordenar una lista.

Python proporciona, al menos, dos formas de ordenar los elementos de una lista:

Conservando la lista original, mediante la función `sorted()` que devuelve una nueva lista ordenada:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> sorted(shopping)
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

Modificando la lista original, mediante la función `sort()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> shopping.sort()
>>> shopping
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

Ambos métodos admiten un parámetro «booleano» `reverse` para indicar si queremos que la ordenación se haga en sentido inverso:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> sorted(shopping, reverse=True)
['Sal', 'Limón', 'Huevos', 'Agua', 'Aceite']
```

Longitud de una lista.

Podemos conocer el número de elementos que tiene una lista con la función `len()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> len(shopping)
5
```

Iterar sobre una lista.

Al igual que hemos visto con las cadenas de texto, también podemos iterar sobre los elementos de una lista utilizando la sentencia for. Todavía no se han explicado los bucles en este manual, pero podemos acceder a esta sección para comprender mejor el funcionamiento de estos ejemplos:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> for product in shopping:
...     print(product)
...
Agua
Huevos
Aceite
Sal
Limón
```

Nota: También es posible usar la sentencia break en este tipo de bucles para abortar su ejecución en algún momento que nos interese.

Iterar usando enumeración.

Hay veces que no sólo nos interesa «visitar» cada uno de los elementos de una lista, sino que también queremos saber su índice dentro de la misma. Para ello Python nos ofrece la función enumerate():

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
>>> for i, product in enumerate(shopping):
...     print(i, product)
...
0 Agua
1 Huevos
2 Aceite
3 Sal
4 Limón
```

Iterar sobre múltiples listas.

Python ofrece la posibilidad de iterar sobre múltiples listas en paralelo utilizando la función zip():

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']
>>> for product, detail in zip(shopping, details):
...     print(product, detail)
...
Agua mineral natural
Aceite de oliva virgen
Arroz basmati
```

Nota: En el caso de que las listas no tengan la misma longitud, la función zip() realiza la combinación hasta que se agota la lista más corta.

Dado que zip() produce un iterador, si queremos obtener una lista explícita con la combinación en paralelo de las listas, debemos construir dicha lista de la siguiente manera:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']
>>> list(zip(shopping, details))
[('Agua', 'mineral natural'),
```

```
('Aceite', 'de oliva virgen'),  
( 'Arroz', 'basmati')]
```

Lista de argumentos de un programa (sys.argv).

Cuando queramos ejecutar un programa Python desde línea de comandos, tendremos la posibilidad de acceder a los argumentos de dicho programa. Para ello se utiliza una lista que la encontramos dentro del módulo sys y que se denomina argv. Veamos un ejemplo:

```
get-args.py  
1 import sys  
2  
3 filename = sys.argv[0]  
4 arg1 = sys.argv[1]  
5 arg2 = float(sys.argv[2])  
6 arg3 = int(sys.argv[3])  
7 arg4 = sys.argv[4]  
8  
9 print(f'{arg1=}')  
10 print(f'{arg2=}')  
11 print(f'{arg3=}')  
12 print(f'{arg4=}')
```

Si lo ejecutamos obtenemos lo siguiente:

```
$ python3 get-args.py hello 99.9 55 "a nice arg"  
arg1='hello'  
arg2=99.9  
arg3=55  
arg4='a nice arg'
```

Funciones matemáticas.

Python nos ofrece, entre otras, estas tres funciones matemáticas básicas que se pueden aplicar sobre listas.

Suma de todos los valores mediante la función sum():

```
>>> data = [5, 3, 2, 8, 9, 1]  
>>> sum(data)  
28
```

Mínimo de todos los valores mediante la función min():

```
>>> data = [5, 3, 2, 8, 9, 1]  
>>> min(data)  
1
```

Máximo de todos los valores mediante la función max():

```
>>> data = [5, 3, 2, 8, 9, 1]  
>>> max(data)  
9
```

Listas de listas.

Como ya hemos visto en varias ocasiones, las listas son estructuras de datos que pueden contener elementos heterogéneos. Estos elementos pueden ser a su vez listas.

A continuación, planteamos un ejemplo con un equipo de fútbol, que suele tener una disposición en el campo organizada en líneas de jugadores. Por ejemplo, en la alineación con la

que España ganó la copa del mundo en 2010 había una disposición 4-3-3 con los siguientes jugadores por línea:

```
>>> goalkeeper = 'Casillas'
>>> defenders = ['Capdevila', 'Piqué', 'Puyol', 'Ramos']
>>> midfielders = ['Xabi', 'Busquets', 'X. Alonso']
>>> forwards = ['Iniesta', 'Villa', 'Pedro']
```

Ahora juntamos las listas en una única lista:

```
>>> team = [goalkeeper, defenders, midfielders, forwards]
>>> team
['Casillas',
 ['Capdevila', 'Piqué', 'Puyol', 'Ramos'],
 ['Xabi', 'Busquets', 'X. Alonso'],
 ['Iniesta', 'Villa', 'Pedro']]
```

Podemos comprobar el acceso a distintos elementos:

```
>>> team[0] # portero
'Casillas'
>>> team[1][0] # lateral izquierdo
'Capdevila'
>>> team[2] # centrocampistas
['Xabi', 'Busquets', 'X. Alonso']
>>> team[3][1] # delantero centro
'Villa'
```

4.2 Tuplas.

El concepto de tupla es muy similar al de lista. Aunque hay algunas diferencias menores, lo fundamental es que, mientras una lista es mutable y se puede modificar, una tupla no admite cambios y, por lo tanto, es inmutable.

4.2.1 Creación de tuplas.

Podemos pensar en crear tuplas tal y como lo hacíamos con listas, pero usando paréntesis en lugar de corchetes:

```
>>> empty_tuple = ()
>>> tenerife_geoloc = (28.46824, -16.25462)
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')
```

Hay que prestar especial atención cuando vamos a crear una tupla de un único elemento. La intención primera sería hacerlo de la siguiente manera:

```
>>> one_item_tuple = ('Papá Noel')
>>> one_item_tuple
'Papá Noel'
>>> type(one_item_tuple)
str
```

Realmente, hemos creado una variable de tipo str (cadena de texto). Para crear una tupla de un elemento debemos añadir una coma al final:

```
>>> one_item_tuple = ('Papá Noel',)
>>> one_item_tuple
('Papá Noel',)
```



```
>>> type(one_item_tuple)
tuple
```

Hay veces que nos podemos encontrar con tuplas que no llevan paréntesis. Quizás no está tan extendido, pero a efectos prácticos tiene el mismo resultado:

```
>>> one_item_tuple = 'Papá Noel',
>>> three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'
>>> tenerife_geoloc = 28.46824, -16.25462
```

4.2.2 Modificación de tuplas.

Como ya hemos comentado previamente, las tuplas son estructuras de datos inmutables. Una vez que las creamos con un valor, no podemos modificarlas. Veamos qué ocurre si lo intentamos:

```
>>> three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'
>>> three_wise_men[0] = 'Tom Hanks'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

4.2.3 Conversión a tuplas.

Para convertir otros tipos de datos en una tupla podemos usar la función `tuple()`:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> tuple(shopping)
('Agua', 'Aceite', 'Arroz')
```

Esta conversión es válida para aquellos tipos de datos que sean iterables: cadenas de caracteres, listas, diccionarios, conjuntos, etc. Por ejemplo, no se podría convertir un número en una tupla.

El uso de la función `tuple()` sin argumentos equivale a crear una tupla vacía:

```
>>> tuple()
()
```

Truco: Para crear una tupla vacía, se suele recomendar el uso de `()` frente a `tuple()`, no sólo por ser más pitónico sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

4.2.4 Operaciones con tuplas.

Con las tuplas podemos realizar todas las operaciones que vimos con listas salvo las que conlleven una modificación «in-situ» de la misma, que son las siguientes:

- ✓ `reverse()`
- ✓ `append()`
- ✓ `extend()`
- ✓ `remove()`
- ✓ `clear()`
- ✓ `sort()`

4.2.5 Desempaquetado de tuplas.

El desempaqueado es una característica de las tuplas que nos permite asignar una tupla a variables independientes. Veamos un ejemplo con código:

```
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')
>>> king1, king2, king3 = three_wise_men
```

```
>>> king1
'Melchor'
>>> king2
'Gaspar'
>>> king3
'Baltasar'
```

Intercambio de valores.

A través del desempaquetado de tuplas podemos llevar a cabo el intercambio de los valores de dos variables de manera directa:

```
>>> value1 = 40
>>> value2 = 20
>>> value1, value2 = value2, value1
>>> value1
20
>>> value2
40
```

Nota: A priori puede parecer que esto es algo «natural», pero en la gran mayoría de lenguajes de programación no es posible hacer este intercambio de forma «directa» ya que necesitamos recurrir a una tercera variable «auxiliar» como almacén temporal en el paso intermedio de traspaso de valores.

Desempaquetado extendido.

No tenemos que ceñirnos a realizar desempaquetado uno a uno. También podemos extenderlo e indicar ciertos «grupos» de elementos mediante el operador *:

```
>>> ranking = ('G', 'A', 'R', 'Y', 'W')
>>> head, *body, tail = ranking
>>> head
'G'
>>> body
['A', 'R', 'Y']
>>> tail
'W'
```

Desempaquetado genérico.

El desempaquetado de tuplas es extensible a cualquier tipo de datos que sea iterable. Veamos algunos ejemplos de ello.

Sobre cadenas de texto:

```
>>> oxygen = 'O2'
>>> first, last = oxygen
>>> first, last
('O', '2')
>>> text = 'Hello, World!'
>>> head, *body, tail = text
>>> head, body, tail
('H', ['e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd'], '!')
```

Sobre listas:

```
>>> writer1, writer2, writer3 = ['Virginia Woolf', 'Jane Austen', 'Mary Shelley']
>>> writer1, writer2, writer3
```

```
(‘Virginia Woolf’, ‘Jane Austen’, ‘Mary Shelley’)
>>> text = ‘Hello, World!’
>>> word1, word2 = text.split()
>>> word1, word2
(‘Hello’, ‘World!’)
```

4.2.6 Tuplas vs Listas.

Aunque puedan parecer estructuras de datos muy similares, sabemos que las tuplas carecen de ciertas operaciones, especialmente las que tienen que ver con la modificación de sus valores, ya que no son inmutables. Si las listas son más flexibles y potentes, ¿por qué íbamos a necesitar tuplas?

Veamos 4 potenciales ventajas del uso de tuplas frente a las listas:

- ✓ Las tuplas ocupan menos espacio en memoria.
- ✓ En las tuplas existe protección frente a cambios indeseados.
- ✓ Las tuplas se pueden usar como claves de diccionarios (son «hashables»).
- ✓ Las namedtuples son una alternativa sencilla a los objetos.

4.3 Diccionarios.

Podemos trasladar el concepto de diccionario de la vida real al de diccionario en Python. Al fin y al cabo, un diccionario es un objeto que contiene palabras, y cada palabra tiene asociado un significado.

Haciendo el paralelismo, diríamos que en Python un diccionario es también un objeto indexado por claves (las palabras) que tienen asociados unos valores (los significados).

Los diccionarios en Python tienen las siguientes características:

- ✓ Mantienen el orden en el que se insertan las claves.
- ✓ Son mutables, con lo que admiten añadir, borrar y modificar sus elementos.
- ✓ Las claves deben ser únicas. A menudo se utilizan las cadenas de texto como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
- ✓ Tienen un acceso muy rápido a sus elementos, debido a la forma en la que están implementados internamente.

Nota: En otros lenguajes de programación, a los diccionarios se les conoce como arrays asociativos, «hashes» o «hashmaps».

4.3.1 Creación de diccionarios.

Para crear un diccionario usamos llaves {} rodeando asignaciones clave: valor que están separadas por comas. Veamos algunos ejemplos de diccionarios:

```
>>> empty_dict = {}
>>> rae = {
... ‘bifronte’: ‘De dos frentes o dos caras’,
... anarcoide: ‘Que tiende al desorden’,
... montuvio: ‘Campesino de la costa’
... }
>>> population_can = {
... 2015: 2_135_209,
```

```
... 2016: 2_154_924,
... 2017: 2_177_048,
... 2018: 2_206_901,
... 2019: 2_220_270
... }
```

En el código anterior, podemos observar la creación de un diccionario vacío, otro donde sus claves y sus valores son cadenas de texto y otro donde las claves y los valores son valores enteros.

4.3.2 Conversión a diccionarios.

Para convertir otros tipos de datos en un diccionario podemos usar la función `dict()`:

```
>>> # Diccionario a partir de una lista de cadenas de texto
>>> dict(['a1', 'b2'])
{'a': '1', 'b': '2'}
>>> # Diccionario a partir de una tupla de cadenas de texto
>>> dict(('a1', 'b2'))
{'a': '1', 'b': '2'}
>>> # Diccionario a partir de una lista de listas
>>> dict(['a', 1], ['b', 2])
{'a': 1, 'b': 2}
```

Nota: Si nos fijamos bien, cualquier iterable que tenga una estructura interna de 2 elementos es susceptible de convertirse en un diccionario a través de la función `dict()`.

Diccionario vacío.

Existe una manera particular de usar `dict()` y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en un diccionario, con lo que obtendremos un diccionario vacío:

```
>>> dict()
{}
```

Truco: Para crear un diccionario vacío, se suele recomendar el uso de `{}` frente a `dict()`, no sólo por ser más pitónico sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

Creación con `dict()`.

También es posible utilizar la función `dict()` para crear diccionarios y no tener que utilizar llaves y comillas. Supongamos que queremos transformar la siguiente tabla en un diccionario:

Atributo	Valor
name	Guido
surname	Van Rossum
job	Python creator

Utilizando la construcción mediante `dict` podemos pasar clave y valor como argumentos de la función:

```
>>> person = dict(
...     name='Guido',
...     surname='Van Rossum',
...     job='Python creator'
... )
>>> person
```

```
{'name': 'Guido', 'surname': 'Van Rossum', 'job': 'Python creator'}
```

El inconveniente que tiene esta aproximación es que las claves deben ser identificadores válidos en Python. Por ejemplo, no se permiten espacios:

```
>>> person = dict(
...     name='Guido van Rossum',
...     date of birth='31/01/1956'
File "<stdin>", line 3
date of birth='31/01/1956'
^
SyntaxError: invalid syntax
```

4.3.3 Operaciones con diccionarios.

Obtener un elemento.

Para obtener un elemento de un diccionario basta con escribir la clave entre corchetes:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }
>>> rae['anarcoide']
'Que tiende al desorden'
```

Si intentamos acceder a una clave que no existe, obtendremos un error:

```
>>> rae['acceso']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'acceso'
```

Usando get().

Existe una función muy útil para «superar» los posibles errores de acceso por claves inexistentes.

Se trata de `get()` y su comportamiento es el siguiente:

- ✓ Si la clave que buscamos existe, nos devuelve su valor.
- ✓ Si la clave que buscamos no existe, nos devuelve `None` salvo que le indiquemos otro valor por defecto, pero en ninguno de los dos casos obtendremos un error.

```
1 >>> rae
2 {'bifronte': 'De dos frentes o dos caras',
3  'anarcoide': 'Que tiende al desorden',
4  'montuvio': 'Campesino de la costa'}
5
6 >>> rae.get('bifronte')
7 'De dos frentes o dos caras'
8
9 >>> rae.get('programación')
10
11 >>> rae.get('programación', 'No disponible')
12 'No disponible'
```

Línea 6: Equivalente a `rae['bifronte']`.

Línea 9: La clave buscada no existe y obtenemos `None`.

Línea 11: La clave buscada no existe y nos devuelve el valor que hemos aportado por defecto.

Añadir o modificar un elemento.

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la clave y asignarle un valor:

- ✓ Si la clave ya existía en el diccionario, se reemplaza el valor existente por el nuevo.
- ✓ Si la clave es nueva, se añade al diccionario con su valor. No vamos a obtener un error a diferencia de las listas.

Partimos del siguiente diccionario para ejemplificar estas acciones:

```
>>> rae = {  
... 'bifronte': 'De dos frentes o dos caras',  
... 'anarcoide': 'Que tiende al desorden',  
... 'montuvio': 'Campesino de la costa'  
... }
```

Vamos a añadir la palabra enjuiciar a nuestro diccionario:

```
>>> rae['enjuiciar'] = 'Someter una cuestión a examen, discusión y juicio'  
>>> rae  
{'bifronte': 'De dos frentes o dos caras',  
'anarcoide': 'Que tiende al desorden',  
'montuvio': 'Campesino de la costa',  
'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'}
```

Supongamos ahora que queremos modificar el significado de la palabra enjuiciar por otra acepción:

```
>>> rae['enjuiciar'] = 'Instruir, juzgar o sentenciar una causa'  
>>> rae  
{'bifronte': 'De dos frentes o dos caras',  
'anarcoide': 'Que tiende al desorden',  
'montuvio': 'Campesino de la costa',  
'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

Creando desde vacío.

Una forma muy habitual de trabajar con diccionarios es utilizar el “patrón de creación” partiendo de uno vacío e ir añadiendo elementos poco a poco.

Supongamos un ejemplo en el que queremos construir un diccionario donde las claves son las letras vocales y los valores son sus posiciones:

```
>>> VOWELS = 'aeiou'  
>>> enum_vowels = {}  
>>> for i, vowel in enumerate(VOWELS):  
...     enum_vowels[vowel] = i + 1  
...  
>>> enum_vowels  
{ 'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5 }
```

Nota: Hemos utilizando la función enumerate() que ya vimos para las listas en el apartado: Iterar usando enumeración.

Pertenencia de una clave.

La forma pitónica de comprobar la existencia de una clave dentro de un diccionario es utilizar el operador in:

```
>>> 'bifronte' in rae
True
>>> 'almohada' in rae
False
>>> 'montuvio' not in rae
False
```

Nota: El operador in siempre devuelve un valor booleano, es decir, verdadero o falso.

Obtener todos los elementos.

Python ofrece mecanismos para obtener todos los elementos de un diccionario. Partimos del siguiente diccionario:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
'anarcoide': 'Que tiende al desorden',
'montuvio': 'Campesino de la costa',
'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

Obtener todas las claves de un diccionario mediante la función keys():

```
>>> rae.keys()
dict_keys(['bifronte', 'anarcoide', 'montuvio', 'enjuiciar'])
```

Obtener todos los valores de un diccionario mediante la función values():

```
>>> rae.values()
dict_values([
'De dos frentes o dos caras',
'Que tiende al desorden',
'Campesino de la costa',
'Instruir, juzgar o sentenciar una causa'
])
```

Obtener todos los pares «clave-valor» de un diccionario mediante la función items():

```
>>> rae.items()
dict_items([
('bifronte', 'De dos frentes o dos caras'),
('anarcoide', 'Que tiende al desorden'),
('montuvio', 'Campesino de la costa'),
('enjuiciar', 'Instruir, juzgar o sentenciar una causa')
])
```

Nota: Para este último caso cabe destacar que los «items» se devuelven como una lista de tuplas, donde cada tupla tiene dos elementos: el primero representa la clave y el segundo representa el valor.

Longitud de un diccionario.

Podemos conocer el número de elementos («clave-valor») que tiene un diccionario con la función len():

```
>>> len(rae)
4
```

Iterar sobre un diccionario.

En función de los elementos que podemos obtener, Python nos proporciona tres maneras de iterar sobre un diccionario.

Iterar sobre claves:

```
>>> for word in rae.keys():
...     print(word)
...
bifronte
anarcoide
montuvio
enjuiciar
```

Iterar sobre valores:

```
>>> for meaning in rae.values():
...     print(meaning)
...
De dos frentes o dos caras
Que tiende al desorden
Campesino de la costa
Instruir, juzgar o sentenciar una causa
```

Iterar sobre «clave-valor»:

```
>>> for word, meaning in rae.items():
...     print(f'{word}: {meaning}')
...
bifronte: De dos frentes o dos caras
anarcoide: Que tiende al desorden
montuvio: Campesino de la costa
enjuiciar: Instruir, juzgar o sentenciar una causa
```

Nota: En este último caso, se ha hecho uso de los «f-strings» para formatear cadenas de texto.

Combinar diccionarios.

Dados dos (o más) diccionarios, es posible «mezclarlos» para obtener una combinación de los mismos. Esta combinación se basa en dos premisas:

- ✓ Si la clave no existe, se añade con su valor.
- ✓ Si la clave ya existe, se añade con el valor del «último» diccionario en la mezcla.

Python ofrece dos mecanismos para realizar esta combinación. Vamos a partir de los siguientes diccionarios para ejemplificar su uso:

```
>>> rae1 = {
... 'bifronte': 'De dos frentes o dos caras',
... 'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'
... }
>>> rae2 = {
... 'anarcoide': 'Que tiende al desorden',
... 'montuvio': 'Campesino de la costa',
... 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'
... }
```


Combinación sin modificar los diccionarios originales, mediante el operador **:

```
>>> {**rae1, **rae2}
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

A partir de Python 3.9 podemos utilizar el operador | para realizar la misma operación que antes:

```
>>> rae1 | rae2
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

Combinación modificando los diccionarios originales, mediante la función update():

```
>>> rae1.update(rae2)
>>> rae1
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

Nota: Debemos tener en cuenta que el orden en el que especificamos los diccionarios a la hora de su combinación (mezcla) es relevante en el resultado final. En este caso el orden de los factores sí altera el producto.

Borrar elementos.

Python nos ofrece, al menos, tres formas para borrar elementos en un diccionario:

Por su clave, mediante la sentencia del:

```
>>> rae = {
... 'bifronte': 'De dos frentes o dos caras',
... 'anarcoide': 'Que tiende al desorden',
... 'montuvio': 'Campesino de la costa'
... }
>>> del(rae['bifronte'])
>>> rae
{'anarcoide': 'Que tiende al desorden', 'montuvio': 'Campesino de la costa'}
```

Por su clave (con extracción): mediante la función pop() podemos extraer un elemento del diccionario por su clave. Vendría a ser una combinación de get() + del:

```
>>> rae.pop('anarcoide')
'Que tiende al desorden'
>>> rae
{'bifronte': 'De dos frentes o dos caras', 'montuvio': 'Campesino de la costa'}
>>> rae.pop('bucle')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'bucle'
```

Advertencia: Si la clave que pretendemos extraer con pop() no existe, obtendremos un error.

Borrado completo del diccionario.

Python ofrece dos posibilidades:

1. Utilizando la función `clear()`:

```
>>> rae = {  
... 'bifronte': 'De dos frentes o dos caras',  
... 'anarcoide': 'Que tiende al desorden',  
... 'montuvio': 'Campesino de la costa'  
... }  
>>> rae.clear()  
>>> rae  
{}
```

2. «Reinicializando» el diccionario a vacío con `{}`:

```
>>> rae = {  
... 'bifronte': 'De dos frentes o dos caras',  
... 'anarcoide': 'Que tiende al desorden',  
... 'montuvio': 'Campesino de la costa'  
... }  
>>> rae = {}  
>>> rae  
{}
```

Nota: La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento.

4.4 Conjuntos.

Un conjunto en Python representa una serie de valores únicos y sin orden establecido, con la única restricción de que sus elementos deben ser «hashables». Mantiene muchas similitudes con el concepto matemático de conjunto.

Se dice que un elemento es hashable si se le puede asignar un valor «hash» que no cambia en ejecución durante toda su vida.

Para crear un conjunto basta con separar sus valores por comas y rodearlos de llaves `{}`:

```
>>> lottery = {21, 10, 46, 29, 31, 94}  
>>> lottery  
{10, 21, 29, 31, 46, 94}
```

No se va a profundizar en el trabajo con conjuntos, puesto que se considera que excede del nivel de conocimientos planeado para este manual.

4.5 Ficheros.

Aunque los ficheros encajarían más en un apartado de «entrada/salida», ya que representan un medio de almacenamiento persistente, también podrían ser vistos como estructuras de datos, puesto que nos permiten guardar la información y asignarles un cierto formato.

Un fichero es un conjunto de bytes almacenados en algún dispositivo. El sistema de ficheros es la estructura lógica que alberga los ficheros y está jerarquizado a través de directorios. Cada fichero se identifica unívocamente a través de una ruta que nos permite acceder a él.

4.5.1 Lectura de un fichero.

Python ofrece la función `open()` para «abrir» un fichero. Esta apertura se puede realizar en 3 modos distintos:

- ✓ Lectura del contenido de un fichero existente.
- ✓ Escritura del contenido en un fichero nuevo.
- ✓ Añadido al contenido de un fichero existente.

Vamos a un ejemplo para leer el contenido de un fichero en el que se encuentran las temperaturas máximas y mínimas de cada día de la última semana. El fichero está en la subcarpeta (ruta relativa) `files/temps.dat` y tiene el siguiente contenido:

```
29 23
31 23
34 26
33 23
29 22
28 22
28 22
```

Lo primero será abrir el fichero:

```
>>> f = open('files/temps.dat')
```

La función `open()` recibe como primer argumento la ruta al fichero que queremos manejar (como un «string») y devuelve el manejador del fichero, que en este caso, lo estamos asignando a una variable llamada `f`, pero le podríamos haber puesto cualquier otro nombre.

Hay que tener en cuenta que la ruta al fichero que abrimos (en modo lectura) debe existir, ya que de lo contrario obtendremos un error:

```
>>> f = open('error.txt')
```

Traceback (most recent call last):

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

FileNotFoundError: [Errno 2] No such file or directory: 'error.txt'

Una vez abierto el fichero ya podemos proceder a leer su contenido. Para ello Python nos ofrece la posibilidad de leer todo el fichero de una vez o bien leerlo línea a línea.

Lectura completa de un fichero.

Para esta operación, Python nos provee, al menos, de dos funciones:

`read()`: devuelve todo el contenido del fichero como una cadena de texto (str):

```
>>> f = open('files/temps.dat')
```

```
>>> f.read()
```

```
'29 23\n31 23\n34 26\n33 23\n29 22\n28 22\n28 22\n'
```

`readlines()`: devuelve todo el contenido del fichero como una lista (list) donde cada elemento es una línea:

```
>>> f = open('files/temps.dat')
```

```
>>> f.readlines()
```

```
['29 23\n', '31 23\n', '34 26\n', '33 23\n', '29 22\n', '28 22\n', '28 22\n']
```

Importante: en ambos casos, los saltos de línea (`\n`) siguen apareciendo en los datos leídos, por lo que habría que limpiar estos caracteres si no quisiéramos que apareciesen.

Lectura línea a línea.

Hay situaciones en las que interesa leer el contenido del fichero línea a línea. Imaginemos un fichero de tamaño considerable (varios GB). Si intentamos leer completamente este fichero de sola una vez podríamos ocupar demasiada RAM y reducir el rendimiento de nuestra máquina.

Es por ello que Python nos ofrece varias aproximaciones a la lectura de ficheros línea a línea. La más usada es iterar sobre el propio manejador del fichero:

```
>>> f = open('files/temps.dat')
>>> for line in f:
...     print(line)
...
29 23
31 23
34 26
33 23
29 22
28 22
28 22
```

Truco: Igual que pasaba anteriormente, la lectura línea por línea también incluye el salto de línea (`\n`). Esto se puede solucionar aplicando `line.split()` para eliminarlo.

4.5.2 Escritura en un fichero.

Para escribir texto en un fichero hay que abrir dicho fichero en modo escritura. Para ello, utilizamos un argumento adicional en la función `open()` que indica esta operación:

```
>>> f = open('files/canary-iata.dat', 'w')
```

Nota: Si bien el fichero en sí mismo se crea al abrirlo en modo escritura, la ruta hasta ese fichero no. Eso quiere decir que debemos asegurarnos de que las carpetas hasta llegar a dicho fichero existen. En otro caso obtenemos un error de tipo `FileNotFoundError`.

Ahora ya podemos hacer uso de la función `write()` para enviar contenido al fichero abierto. Supongamos que queremos volcar el contenido de una lista en dicho fichero. En este caso usaremos los códigos IATA de aeropuertos de las Islas Canarias.

```
1 >>> canary_iata = ("GCFV", "GCHI", "GCLA", "GCLP", "GCGM", "GCRR", "GCTS", "GCXO")
2
3 >>> for code in canary_iata:
4 ...     f.write(code + '\n')
5 ...
6
7 >>> f.close()
```

Aclaraciones:

- ✓ Línea 4: escritura de cada código en el fichero. La función `write()` no incluye el salto de línea por defecto, así que lo añadimos de manera explícita.
- ✓ Línea 7: cierre del fichero con la función `close()`. Especialmente en el caso de la escritura de ficheros, se recomienda encarecidamente cerrar los ficheros para evitar pérdida de datos.

Advertencia: Siempre que se abre un fichero en modo escritura utilizando el argumento `'w'`, el fichero se inicializa, borrando cualquier contenido que pudiera tener.

4.5.3 Añadido a un fichero.

La única diferencia entre añadir información a un fichero y escribir información en un fichero es el modo de apertura del fichero. En este caso utilizamos 'a' por «append»:

```
>>> f = open('more-data.txt', 'a')
```

En este caso, el fichero more-data.txt se abrirá en modo añadir, con lo que las llamadas a la función write() hará que aparezcan nueva información al final del contenido ya existente en dicho fichero.

4.5.4 Usando contextos.

Python ofrece gestores de contexto como una solución para establecer reglas de entrada y salida a un determinado bloque de código.

En el caso que nos ocupa, usaremos la sentencia with y el contexto creado se ocupará de cerrar adecuadamente el fichero que hemos abierto, liberando así sus recursos:

```
1 >>> with open('files/temps.dat') as f:
2 ...     for line in f:
3 ...         max_temp, min_temp = line.strip().split()
4 ...         print(max_temp, min_temp)
5 ...
6 29 23
7 31 23
8 34 26
9 33 23
10 29 22
11 28 22
12 28 22
```

Aclaraciones:

- ✓ Línea 1: apertura del fichero en modo lectura utilizando el gestor de contexto definido por la palabra reservada with.
- ✓ Línea 2: lectura del fichero línea a línea utilizando la iteración sobre el manejador del fichero.
- ✓ Línea 3: limpieza de saltos de línea con strip() encadenando la función split() para separar las dos temperaturas por el carácter espacio.
- ✓ Línea 4: imprimir por pantalla la temperatura mínima y la máxima.

Nota: Es una buena práctica usar with cuando se manejan ficheros. La ventaja es que el fichero se cierra adecuadamente en cualquier circunstancia, incluso si se produce cualquier tipo de error.

Hay que prestar atención a la hora de escribir valores numéricos en un fichero, ya que el método write() por defecto espera ver un «string» como argumento:

```
>>> lottery = [43, 21, 99, 18, 37, 99]
>>> with open('files/lottery.dat', 'w') as f:
...     for number in lottery:
...         f.write(number + '\n')
...
Traceback (most recent call last):
File "<stdin>", line 3, in <module>
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Importante: para evitar este tipo de errores, se debe convertir a str aquellos valores que queramos usar con la función write() para escribir información en un fichero de texto.

5. Estructuras de control de flujo.

Todo programa informático está formado por instrucciones que se ejecutan en forma secuencial de «arriba» a «abajo», de igual manera que leeríamos un libro. Este orden constituye el llamado flujo del programa.

Es posible modificar este flujo secuencial para que tome bifurcaciones o repita ciertas instrucciones. Las sentencias que nos permiten hacer estas modificaciones se engloban en el control de flujo.

5.1 Aspectos previos.

Antes de conocer las estructuras de control de flujo, introduciremos algunas cuestiones generales de escritura de código.

5.1.1 Definición de bloques.

A diferencia de otros lenguajes que utilizan llaves para definir los bloques de código, cuando Guido Van Rossum creó Python quiso evitar estos caracteres por considerarlos innecesarios. Es por ello que en este lenguaje los bloques de código se definen a través de espacios en blanco, (preferiblemente 4). En términos técnicos se habla del tamaño de indentación.

5.1.2 Comentarios.

Los comentarios son anotaciones que podemos incluir en nuestro programa y que nos permiten aclarar ciertos aspectos del código. Estas indicaciones son ignoradas por el intérprete de Python.

Los comentarios se incluyen usando el símbolo almohadilla # y comprenden hasta el final de la línea.

```
# Universe age expressed in days
universe_age = 13800 * (10 ** 6) * 365
```

Los comentarios también pueden aparecer en la misma línea de código, aunque la guía de estilo de Python no aconseja usarlos en demasía:

```
stock = 0      # Release additional articles
```

5.1.3 Ancho del código.

Los programas suelen ser más legibles cuando las líneas no son excesivamente largas. La longitud máxima de línea recomendada por la guía de estilo de Python es de 80 caracteres.

Sin embargo, esto genera una cierta polémica hoy en día, ya que los tamaños de pantalla han aumentado y las resoluciones son mucho mayores que hace años. Así las líneas de más de 80 caracteres se siguen visualizando correctamente. Hay personas que son más estrictas en este límite y otras más flexibles.

En caso de que queramos romper una línea de código demasiado larga, tenemos dos opciones:

1. Usar la barra invertida \:

```
>>> factorial = 4 * 3 * 2 * 1
>>> factorial = 4 * \
... 3 * \
... 2 * \
```

... 1

2. Usar los paréntesis (...):

```
>>> factorial = 4 * 3 * 2 * 1
>>> factorial = (4 *
... 3 *
... 2 *
... 1)
```

5.2 Condicionales.

En esta sección veremos las sentencias “if” y “match-case” junto a las distintas variantes que pueden asumir.

5.2.1 La sentencia if.

La sentencia condicional en Python (al igual que en muchos otros lenguajes de programación) es if. En su escritura debemos añadir una expresión de comparación terminando con dos puntos (:) al final de la línea. Veamos un ejemplo:

```
>>> temperature = 40
>>> if temperature > 35:
...     print('Aviso por alta temperatura')
...
Aviso por alta temperatura
```

Nota: en Python no es necesario incluir paréntesis “()” al escribir condiciones. Sin embargo, hay veces que es recomendable por claridad o por establecer prioridades.

En el caso anterior se puede ver claramente que la condición se cumple y por tanto se ejecuta la instrucción que tenemos dentro del cuerpo de la condición. Pero podría no ser así. Para controlar ese caso existe la sentencia else:

```
>>> temperature = 20
>>> if temperature > 35:
...     print('Aviso por alta temperatura')
... else:
...     print('Parámetros normales')
...
Parámetros normales
```

Podríamos tener incluso condiciones dentro de condiciones, lo que se viene a llamar técnicamente condiciones anidadas:

```
>>> temperature = 28
>>> if temperature < 20:
...     if temperature < 10:
...         print('Nivel azul')
...     else:
...         print('Nivel verde')
... else:
...     if temperature < 30:
...         print('Nivel naranja')
...     else:
...         print('Nivel rojo')
```



```
...  
Nivel naranja
```

Python nos ofrece una mejora en la escritura de condiciones anidadas cuando aparecen consecutivamente un else y un if, ya que podemos sustituirlos por la sentencia elif. Aplicaremos esta mejora al código del ejemplo anterior:

```
>>> temperature = 28  
>>> if temperature < 20:  
...     if temperature < 10:  
...         print('Nivel azul')  
...     else:  
...         print('Nivel verde')  
... elif temperature < 30:  
...     print('Nivel naranja')  
... else:  
...     print('Nivel rojo')  
...  
Nivel naranja
```

5.2.2 Operadores de comparación.

Cuando escribimos condiciones debemos incluir alguna expresión de comparación. Para usar estas expresiones es fundamental conocer los operadores que nos ofrece Python:

Operador	Símbolo
Igualdad	==
Desigualdad	!=
Menor que	<
Menor o igual que	<=
Mayor que	>
Mayor o igual que	>=

A continuación, vamos a ver una serie de ejemplos con expresiones de comparación. Tengamos en cuenta que estas expresiones habría que incluirlas dentro de la sentencia condicional en el caso de que quisiéramos tomar una acción concreta:

```
# Asignación de valor inicial
```

```
>>> value = 8  
>>> value == 8  
True  
>>> value != 8  
False  
>>> value < 12  
True  
>>> value <= 7  
False  
>>> value > 4  
True  
>>> value >= 9  
False
```

Podemos escribir condiciones más complejas usando los operadores lógicos:

- ✓ and
- ✓ or
- ✓ not

```
# Asignación de valor inicial
>>> x = 8
>>> x > 4 or x > 12 # True or False
True
>>> x < 4 or x > 12 # False or False
False
>>> x > 4 and x > 12 # True and False
False
>>> x > 4 and x < 12 # True and True
True
>>> not(x != 8) # not False
True
```

Python ofrece la posibilidad de ver si un valor está entre dos límites de manera directa. Así, por ejemplo, para descubrir si value está entre 4 y 12 haríamos:

```
>>> 4 <= value <= 12
True
```

Notas:

- ✓ Una expresión de comparación siempre devuelve un valor booleano, es decir True o False.
- ✓ El uso de paréntesis, en función del caso, puede aclarar la expresión de comparación.

Variables y valores booleanos» en condiciones.

Cuando queremos preguntar por la veracidad de una determinada variable «booleana» en una condición, la primera aproximación que parece razonable es la siguiente:

```
>>> is_cold = True
>>> if is_cold == True:
...     print('Coge chaqueta')
... else:
...     print('Usa camiseta')
...
Coge chaqueta
```

Pero podemos simplificar esta condición de la siguiente forma:

```
>>> if is_cold:
...     print('Coge chaqueta')
... else:
...     print('Usa camiseta')
...
Coge chaqueta
```

Hemos visto una comparación para un valor «booleano» verdadero (True). En el caso de que la comparación fuera para un valor falso lo haríamos así:

```
>>> is_cold = False
>>> if not is_cold: # Equivalente a if is_cold == False
...     print('Usa camiseta')
... else:
...     print('Coge chaqueta')
...
Usa camiseta
```

De hecho, si lo pensamos, estamos reproduciendo bastante bien el lenguaje natural:

- ✓ Si hace frío, coge chaqueta.
- ✓ Si no hace frío, usa camiseta.

Valor nulo.

“None” es un valor especial de Python que almacena el valor nulo. Veamos cómo se comporta al incorporarlo en condiciones de veracidad:

```
>>> value = None
>>> if value:
...     print('Value has some useful value')
... else:
...     # value podría contener None, False (u otro valor diferente)
...     print('Value seems to be void')
...
Value seems to be void
```

Como se aprecia, podríamos confundir el valor False (u otros) con el valor None. Para distinguir None de los valores propiamente booleanos, se recomienda el uso del operador is:

```
>>> value = None
>>> if value is None:
...     print('Value is clearly None')
... else:
...     # value podría contener True, False (u otro)
...     print('Value has some useful value')
...
Value is clearly void
```

De igual forma, podemos usar esta construcción para el caso contrario. La forma «pitónica» de preguntar si algo no es nulo es la siguiente:

```
>>> value = 99
>>> if value is not None:
...     print(f'{value=}')
...
value=99
```

5.2.3 Sentencia match-case.

Una de las novedades más esperadas (y quizás controvertidas) de Python 3.10 fue el llamado Structural Pattern Matching que introdujo en el lenguaje una nueva sentencia condicional. Ésta se podría asemejar a la sentencia «switch» que ya existe en otros lenguajes de programación.

Comparando valores.

En su versión más simple, el «pattern matching» permite comparar un valor de entrada con una serie de literales. Algo así como un conjunto de sentencias «if» encadenadas:

```
>>> color = '#FF0000'
>>> match color:
...     case '#FF0000':
...         print('Rojo')
...     case '#00FF00':
...         print('Verde')
...     case '#0000FF':
...         print('Azul')
```

```
...
Rojo
```

¿Qué ocurre si el valor que comparamos no existe entre las opciones disponibles? Pues en principio, nada, ya que este caso no está cubierto. Si lo queremos controlar, hay que añadir una nueva regla utilizando el subguión `_` como patrón:

```
>>> color = '#FF0000'
>>> match color:
...     case '#FF0000':
...         print('Rojo')
...     case '#00FF00':
...         print('Verde')
...     case '#0000FF':
...         print('Azul')
...     case _:
...         print('Color desconocido!')
...
Color desconocido!
```

Patrones avanzados.

La sentencia `match-case` va mucho más allá de una simple comparación de valores. Con ella podremos deconstruir estructuras de datos, capturar elementos o mapear valores.

Para ejemplificar varias de sus funcionalidades, vamos a partir de una tupla que representará un punto en el plano (2 coordenadas) o en el espacio (3 coordenadas). Lo primero que vamos a hacer es detectar en qué dimensión se encuentra el punto:

```
>>> point = (2, 5)
>>> match point:
...     case (x, y):
...         print(f'({x},{y}) is in plane')
...     case (x, y, z):
...         print(f'({x},{y},{z}) is in space')
...
(2,5) is in plane
```

```
>>> point = (3, 1, 7)
>>> match point:
...     case (x, y):
...         print(f'({x},{y}) is in plane')
...     case (x, y, z):
...         print(f'({x},{y},{z}) is in space')
...
(3,1,7) is in space
```

Este mismo ejemplo lo podríamos usar con un punto formado por «strings»:

```
>>> point = ('2', '5')
```

Por lo tanto, en un siguiente paso, vamos a restringir nuestros patrones a valores enteros:

```
>>> point = ('2', '5')
>>> match point:
...     case (int(), int()):
...         print(f'{point} is in plane')
```

```

...     case (int(), int(), int()):
...         print(f'{point} is in space')
...     case _:
...         print('Unknown!')
...
Unkonwn!

```

Imaginemos ahora que nos piden calcular la distancia del punto al origen. Debemos tener en cuenta que, a priori, desconocemos si el punto está en el plano o en el espacio:

```

>>> point = (8, 3, 5)
>>> match point:
...     case (int(x), int(y)):
...         dist_to_origin = (x ** 2 + y ** 2) ** (1 / 2)
...     case (int(x), int(y), int(z)):
...         dist_to_origin = (x ** 2 + y ** 2 + z ** 2) ** (1 / 2)
...     case _:
...         print('Unknown!')
...
>>> dist_to_origin
9.899494936611665

```

De esta manera, nos aseguramos que los puntos de entrada deben tener todas sus coordenadas como valores enteros. Si tuviéramos un caso como el siguiente, devolvería 'Unknown!':

```

>>> point = ('8', 3, 5) # Nótese el 8 como "string"

```

En este otro ejemplo veremos un fragmento de código en el que tenemos que comprobar la estructura de un bloque de autenticación definido mediante un diccionario. Los métodos válidos de autenticación son únicamente dos: bien usando nombre de usuario y contraseña, o bien usando correo electrónico y «token» de acceso. Además, los valores deben venir en formato cadena de texto:

```

1 >>> # Lista de diccionarios
2 >>> auths = [
3 ...     {'username': 'sdelquin', 'password': '1234'},
4 ...     {'email': 'sdelquin@gmail.com', 'token': '4321'},
5 ...     {'email': 'test@test.com', 'password': 'ABCD'},
6 ...     {'username': 'sdelquin', 'password': '1234'}
7 ... ]
8
9 >>> for auth in auths:
10 ...     print(auth)
11 ...     match auth:
12 ...         case {'username': str(username), 'password': str(password)}:
13 ...             print('Authenticating with username and password')
14 ...             print(f'{username}: {password}')
15 ...         case {'email': str(email), 'token': str(token)}:
16 ...             print('Authenticating with email and token')
17 ...             print(f'{email}: {token}')
18 ...         case _:
19 ...             print('Authenticating method not valid!')
20 ...             print('---')
21 ...
22 {'username': 'sdelquin', 'password': '1234'}

```

```

23 Authenticating with username and password
24 sdelquin: 1234
25 ---
26 {'email': 'sdelquin@gmail.com', 'token': '4321'}
27 Authenticating with email and token
28 sdelquin@gmail.com: 4321
29 ---
30 {'email': 'test@test.com', 'password': 'ABCD'}
31 Authenticating method not valid!
32 ---
33 {'username': 'sdelquin', 'password': 1234}
34 Authenticating method not valid!
35 ---

```

En un último ejemplo veremos un código que nos indica si, dada la edad de una persona, puede beber alcohol:

```

1 >>> age = 21
2
3 >>> match age:
4 ...     case 0 | None:
5 ...         print('Not a person')
6 ...     case n if n < 17:
7 ...         print('Nope')
8 ...     case n if n < 22:
9 ...         print('Not in the US')
10 ...     case _:
11 ...         print('Yes')
12 ...
13 Not in the US

```

Aclaraciones:

- ✓ En la línea 4 podemos observar el uso del operador OR.
- ✓ En las líneas 6 y 8 podemos observar el uso de condiciones dando lugar a “cláusulas guarda”.

5.3 Bucles.

Cuando queremos hacer algo más de una vez, necesitamos recurrir a un bucle. En esta sección veremos las distintas sentencias en Python que nos permiten repetir un bloque de código.

5.3.1 La sentencia *while*.

El primer mecanismo que existe en Python para repetir instrucciones es usar la sentencia *while*. La semántica tras esta sentencia es: «Mientras se cumpla la condición haz algo». Veamos un sencillo bucle que muestra por pantalla los números del 1 al 4:

```

>>> value = 1
>>> while value <= 3:
...     print(value)
...     value += 1
...
1
2

```

3
4

La condición del bucle se comprueba en cada nueva repetición. En este caso chequeamos que la variable `value` sea menor o igual que 4. Dentro del cuerpo del bucle estamos incrementando esa variable en 1 unidad.

Romper un bucle `while`.

Python ofrece la posibilidad de romper o finalizar un bucle antes de que se cumpla la condición de parada. Supongamos un ejemplo en el que estamos buscando el primer número múltiplo de 3 yendo desde 20 hasta 1:

```
>>> num = 20
>>> while num >= 1:
...     if num % 3 == 0:
...         print(num)
...         break
...     num -= 1
...
18
```

Como hemos visto en este ejemplo, `break` nos permite finalizar el bucle una vez que hemos encontrado nuestro objetivo: el primer múltiplo de 3. Pero si no lo hubiéramos encontrado, el bucle habría seguido decrementando la variable `num` hasta valer 0, y la condición del bucle `while` hubiera resultado falsa.

Comprobar la rotura.

Python nos ofrece la posibilidad de detectar si el bucle ha acabado de forma ordinaria, es decir, que ha finalizado por no cumplirse la condición establecida. Para ello podemos hacer uso de la sentencia `else` como parte del propio bucle.

Si el bucle `while` finaliza normalmente (sin llamada a `break`) el flujo de control pasa a la sentencia opcional `else`. Veamos un ejemplo en el que tratamos de encontrar un múltiplo de 9 en el rango [1, 8] (es obvio que no sucederá):

```
>>> num = 8
>>> while num >= 1:
...     if num % 9 == 0:
...         print(f'{num} is a multiple of 9!')
...         break
...     num -= 1
... else:
...     print('No multiples of 9 found!')
...
No multiples of 9 found!
```

Continuar un bucle.

Hay situaciones en las que, en vez de romper un bucle, nos interesa saltar adelante hacia la siguiente repetición. Para ello Python nos ofrece la sentencia `continue` que hace precisamente eso, descartar el resto del código del bucle y saltar a la siguiente iteración.

Veamos un ejemplo en el que usaremos esta estrategia para mostrar todos los números en el rango [1, 20], ignorando aquellos que sean múltiplos de 3:

```
>>> num = 21
>>> while num >= 1:
...     num -= 1
...     if num % 3 == 0:
...         continue
...     print(num, end=', ') # Evitar salto de línea
...
20, 19, 17, 16, 14, 13, 11, 10, 8, 7, 5, 4, 2, 1,
```

5.3.2 La sentencia *for*.

Python permite recorrer aquellos tipos de datos que sean iterables, es decir, que admitan iterar sobre ellos. Algunos ejemplos de tipos y estructuras de datos que permiten ser iteradas (recorridas) son: cadenas de texto, listas, diccionarios, ficheros, etc. La sentencia *for* nos permite realizar esta acción. Ya se han visto diferentes ejemplos en otras secciones anteriores a esta.

A continuación, se plantea un ejemplo en el que vamos a recorrer (iterar) una cadena de texto:

```
>>> word = 'Python'
>>> for letter in word:
...     print(letter)
...
P
y
t
h
o
n
```

La clave aquí está en darse cuenta de que el bucle va tomando, en cada iteración, cada uno de los elementos de la variable que especifiquemos. En este caso concreto *letter* va tomando cada una de las letras que existen en *word*, porque una cadena de texto está formada por elementos que son caracteres.

Importante: La variable que utilizamos en el bucle *for* para ir tomando los valores puede tener cualquier nombre. Al fin y al cabo, es una variable que definimos según nuestras necesidades. Debemos tener en cuenta que se suele usar un nombre en singular.

Romper un bucle *for*.

Una sentencia *break* dentro de un *for* rompe el bucle igual que veíamos para los bucles *while*. Veamos un ejemplo con el código anterior. En este caso vamos a recorrer una cadena de texto y pararemos el bucle cuando encontremos una letra *t* minúscula:

```
>>> word = 'Python'
>>> for letter in word:
...     if letter == 't':
...         break
...     print(letter)
...
P
y
```


Truco: Tanto la “comprobación de rotura” de un bucle como la “continuación” a la siguiente iteración se llevan a cabo del mismo modo que hemos visto con los bucles de tipo while.

Secuencias de números.

Es muy habitual hacer uso de secuencias de números en bucles. Python no tiene una instrucción específica para ello. Lo que sí aporta es una función `range()` que devuelve un flujo de números en el rango especificado. Una de las grandes ventajas es que la «lista» generada no se construye explícitamente, sino que cada valor se genera bajo demanda. Esta técnica mejora el consumo de recursos, especialmente en términos de memoria.

La técnica para la generación de secuencias de números es muy similar a la utilizada en los «slices» de cadenas de texto. En este caso disponemos de la función `range(start, stop, step)`:

- ✓ `start`: Es opcional y tiene valor por defecto 0.
- ✓ `stop`: es obligatorio (siempre se llega a 1 menos que este valor).
- ✓ `step`: es opcional y tiene valor por defecto 1.

`range()` devuelve un objeto iterable, así que iremos obteniendo los valores paso a paso con una sentencia `for ... in`. Veamos diferentes ejemplos de uso:

Rango: [0, 1, 2]

```
>>> for i in range(0, 3):
```

```
... print(i)
```

```
...
```

```
0
```

```
1
```

```
2
```

```
>>> for i in range(3):
```

```
... print(i)
```

```
...
```

```
0
```

```
1
```

```
2
```

Rango: [1, 3, 5]

```
>>> for i in range(1, 6, 2):
```

```
... print(i)
```

```
...
```

```
1
```

```
3
```

```
5
```

Rango: [2, 1, 0]

```
>>> for i in range(2, -1, -1):
```

```
... print(i)
```

```
...
```

```
2
```

```
1
```

```
0
```

Truco: Se suelen utilizar nombres de variables `i`, `j`, `k` para lo que se denominan contadores. Este tipo de variables toman valores numéricos enteros como en los ejemplos anteriores. No conviene generalizar el uso de estas variables a situaciones en las que, claramente, tenemos la

posibilidad de asignar un nombre semánticamente más significativo. Esto viene de tiempos antiguos en FORTRAN donde i era la primera letra que tenía valor entero por defecto.

Uso del guión bajo.

Hay situaciones en las que no necesitamos usar la variable que toma valores en el rango, sino que únicamente queremos repetir una acción un número determinado de veces.

Para estos casos se suele recomendar usar el guión bajo `_` como nombre de variable, que da a entender que no estamos usando esta variable de forma explícita:

```
>>> for _ in range(10):
...     print('Repeat me 10 times!')
...
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
```

5.3.3 Bucles anidados.

Como ya vimos en las sentencias condicionales, el anidamiento es una técnica por la que incluimos distintos niveles de encapsulamiento de sentencias, unas dentro de otras, con mayor nivel de profundidad. En el caso de los bucles también es posible hacer anidamiento.

Veamos un ejemplo de 2 bucles anidados en el que generamos todas las tablas de multiplicar:

```
>>> for i in range(1, 10):
...     for j in range(1, 10):
...         result = i * j
...     print(f'{i} * {j} = {result}')
...
```

Los resultados irán desde “1 x 1 = 1” hasta “9 x 9 = 81”.

Lo que está ocurriendo en este código es que, para cada valor que toma la variable `i`, la otra variable `j` toma todos sus valores. Como resultado tenemos una combinación completa de los valores en el rango especificado.

Notas:

- ✓ Podemos añadir todos los niveles de anidamiento que queramos. Eso sí, hay que tener en cuenta que cada nuevo nivel de anidamiento supone un importante aumento de la complejidad ciclomática de nuestro código, lo que se traduce en mayores tiempos de ejecución.
- ✓ Los bucles anidados también se pueden aplicar en la sentencia `while`.

6. Modularidad.

La modularidad es la característica de un sistema que permite que sea estudiado, visto o entendido como la unión de varias partes que interactúan entre sí y que trabajan solidariamente para alcanzar un objetivo común, realizando cada una de ellas una tarea necesaria para la consecución de dicho objetivo.

Cada una de esas partes en que se encuentre dividido el sistema recibe el nombre de módulo. Idealmente, un módulo debe poder cumplir las condiciones de caja negra, es decir, ser independiente del resto de los módulos y comunicarse con ellos (con todos o sólo con una parte) a través de entradas y salidas bien definidas.

En esta sección veremos las facilidades que nos proporciona Python para trabajar la modularidad del código.

6.1 Funciones.

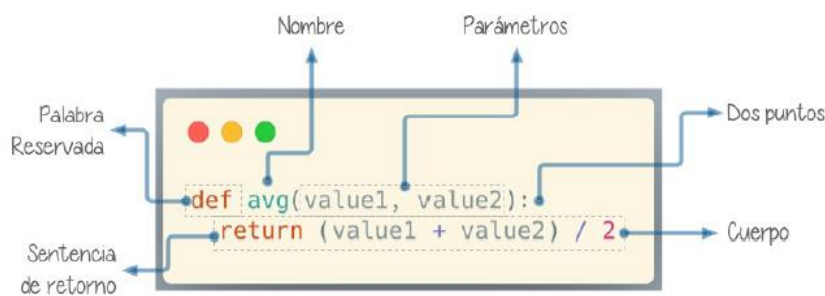
El concepto de función es básico en prácticamente cualquier lenguaje de programación. Se trata de una estructura que nos permite agrupar código. Persigue dos objetivos claros:

- ✓ No repetir trozos de código durante nuestro programa.
- ✓ Reutilizar el código para distintas situaciones.

Una función viene definida por su nombre, sus parámetros y su valor de retorno. Esta parametrización de las funciones las convierte en una poderosa herramienta ajustable a las circunstancias que tengamos. Al invocarla estaremos solicitando su ejecución y obtendremos unos resultados.

6.1.1 Definir una función.

Para definir una función utilizamos la palabra reservada `def` seguida del nombre de la función. A continuación, aparecerán 0 o más parámetros separados por comas (entre paréntesis), finalizando la línea con dos puntos (:). En la siguiente línea empezaría el cuerpo de la función que puede contener 1 o más sentencias, incluyendo (o no) una sentencia de retorno del resultado mediante `return`.



Advertencia: debemos prestar especial atención a los dos puntos (:) porque suelen olvidarse en la definición de la función.

Hagamos una primera función sencilla que no recibe parámetros:

```
def say_hello():  
    print('Hello!')
```

- ✓ Nótese la indentación (sangrado) del cuerpo de la función.
- ✓ Los nombres de las funciones siguen las mismas reglas que las variables.

Invocar una función.

Para invocar (o «llamar») a una función sólo tendremos que escribir su nombre seguido de paréntesis. En el caso de la función sencilla (vista anteriormente) se haría así:

```
>>> def say_hello():
...     print('Hello!')
...
>>> say_hello()
Hello!
```

Como era de esperar, al invocar a esta función obtenemos un mensaje por pantalla, fruto de la ejecución del cuerpo de la función.

Retornar un valor.

Las funciones pueden retornar (o «devolver») un valor. Veamos un ejemplo muy sencillo:

```
>>> def one():
...     return 1
...
>>> one()
1
```

Importante: no debemos confundir return con print(). El valor de retorno de una función nos permite usarlo fuera de su contexto. El hecho de añadir print() al cuerpo de una función es algo «coyuntural» y no modifica el resultado de la lógica interna.

Nota: en la sentencia return podemos incluir variables y expresiones, no únicamente literales.

Pero no sólo podemos invocar a la función directamente, sino que también la podemos integrar en otras expresiones. Por ejemplo, en condicionales:

```
>>> if one() == 1:
...     print('It works!')
... else:
...     print('Something is broken')
...
It works!
```

Si una función no incluye un return de forma explícita, devolverá None de forma implícita:

```
>>> def empty():
...     x = 0
...
>>> print(empty())
None
```

6.1.2 Parámetros y argumentos.

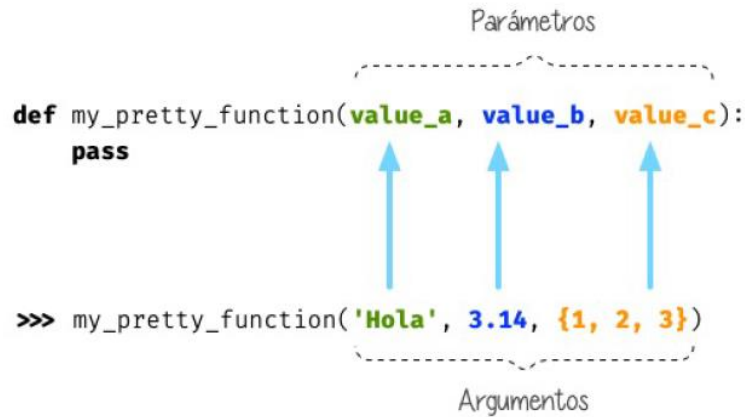
Si una función no dispusiera de valores de entrada estaría muy limitada en su actuación. Es por ello que los parámetros nos permiten variar los datos que consume una función para obtener distintos resultados. Vamos a empezar a crear funciones que reciben parámetros.

En este caso escribiremos una función que recibe un valor numérico y devuelve su raíz cuadrada:

```
>>> def sqrt(value):
...     return value ** (1/2)
```

```
...
>>> sqrt(4)
2.0
```

Cuando llamamos a una función con argumentos, los valores de estos argumentos se copian en los correspondientes parámetros dentro de la función:



Truco: La sentencia pass permite «no hacer nada». Es una especie de «placeholder».

Veamos otra función con dos parámetros y algo más de lógica de negocio:

```
>>> def _min(a, b):
...     if a < b:
...         return a
...     else:
...         return b
...
>>> _min(7, 9)
7
```

Argumentos posicionales.

Los argumentos posicionales son aquellos argumentos que se copian en sus correspondientes parámetros en orden. Vamos a mostrar un ejemplo definiendo una función que construye una «cpu» a partir de 3 parámetros:

```
>>> def build_cpu(vendor, num_cores, freq):
...     return dict(
...         vendor=vendor,
...         num_cores=num_cores,
...         freq=freq
...     )
...
```

Una posible llamada a la función con argumentos posicionales sería la siguiente:

```
>>> build_cpu('AMD', 8, 2.7)
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Lo que ha sucedido es un mapeo directo entre argumentos y parámetros en el mismo orden que estaban definidos:

Parámetro	Argumento
vendor	AMD
num_cores	8
freq	2.7

Pero es evidente que una clara desventaja del uso de argumentos posicionales es que se necesita recordar el orden de los argumentos. Un error en la posición de los argumentos puede causar resultados indeseados:

```
>>> build_cpu(8, 2.7, 'AMD')
{'vendor': 8, 'num_cores': 2.7, 'freq': 'AMD'}
```

Argumentos nominales.

En esta aproximación, los argumentos no son copiados en un orden específico, sino que se asignan por nombre a cada parámetro. Ello nos permite salvar el problema de conocer cuál es el orden de los parámetros en la definición de la función. Para utilizarlo, basta con realizar una asignación de cada argumento en la propia llamada a la función.

Veamos la misma llamada que hemos hecho en el ejemplo de construcción de la «cpu» pero ahora utilizando paso de argumentos nominales:

```
>>> build_cpu(vendor='AMD', num_cores=8, freq=2.7)
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Se puede ver claramente que el orden de los argumentos no influye en el resultado final:

```
>>> build_cpu(num_cores=8, freq=2.7, vendor='AMD')
{'vendor': 'AMD', 'num_cores': 8, 'freq': 2.7}
```

Argumentos posicionales y nominales.

Python permite mezclar argumentos posicionales y nominales en la llamada a una función:

```
>>> build_cpu('INTEL', num_cores=4, freq=3.1)
{'vendor': 'INTEL', 'num_cores': 4, 'freq': 3.1}
```

Pero hay que tener en cuenta que, en este escenario, los argumentos posicionales siempre deben ir antes que los argumentos nominales. Esto tiene mucho sentido ya que, de no hacerlo así, Python no tendría forma de discernir a qué parámetro corresponde cada argumento:

```
>>> build_cpu(num_cores=4, 'INTEL', freq=3.1)
File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Parámetros por defecto.

Es posible especificar valores por defecto en los parámetros de una función. En el caso de que no se proporcione un valor al argumento en la llamada a la función, el parámetro correspondiente tomará el valor definido por defecto.

Siguiendo con el ejemplo de la «cpu», podemos asignar 2.0GHz como frecuencia por defecto.

La definición de la función cambiaría ligeramente:

```
>>> def build_cpu(vendor, num_cores, freq=2.0):
...     return dict(
...         vendor=vendor,
...         num_cores=num_cores,
...         freq=freq
...     )
...
```

Esta sería la llamada a la función sin especificar frecuencia de «cpu»:

```
>>> build_cpu('INTEL', 2)
{'vendor': 'INTEL', 'num_cores': 2, 'freq': 2.0}
```

Esta sería la llamada a la función indicando una frecuencia concreta de «cpu»:

```
>>> build_cpu('INTEL', 2, 3.4)
{'vendor': 'INTEL', 'num_cores': 2, 'freq': 3.4}
```

Importante: los valores por defecto en los parámetros se calculan cuando se define la función, no cuando se ejecuta.

Modificando parámetros mutables.

Hay que tener cuidado a la hora de manejar los parámetros que pasamos a una función ya que podemos obtener resultados indeseados, especialmente cuando trabajamos con tipos de datos mutables.

Supongamos una función que añada elementos a una lista que pasamos por parámetro. La idea es que, si no pasamos la lista, ésta siempre empiece siendo vacía. Hagamos una serie de pruebas pasando alguna lista como segundo argumento:

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...
>>> buggy('a', [])
['a']
>>> buggy('a', ['x', 'y', 'z'])
['x', 'y', 'z', 'a']
```

Aparentemente todo está funcionando de manera correcta, pero veamos qué ocurre en las siguientes llamadas:

```
>>> buggy('a')
['a']
>>> buggy('b') # Se esperaría ['b']
['a', 'b']
```

Obviamente algo no ha funcionado correctamente. Se esperaría que result tuviera una lista vacía en cada ejecución. Sin embargo, esto no sucede por estas dos razones:

- ✓ El valor por defecto se establece cuando se define la función.
- ✓ La variable result apunta a una zona de memoria en la que se modifican sus valores.

La forma de solucionar el problema sería utilizando un parámetro con valor por defecto que tenga un tipo de dato inmutable, y tener en cuenta cuál es la primera llamada:

```
>>> def nonbuggy(arg, result=None):
...     if result is None:
...         result = []
...     result.append(arg)
...     print(result)
...
>>> nonbuggy('a')
['a']
>>> nonbuggy('b')
['b']
```

Empaquetar/Desempaquetar argumentos.

Python nos ofrece la posibilidad de empaquetar y desempaquetar argumentos cuando estamos invocando a una función, tanto para argumentos posicionales como para argumentos nominales.

Y de este hecho se deriva que podamos utilizar un número variable de argumentos en una función, algo que puede ser muy interesante según el caso de uso que tengamos.

Empaquetar/Desempaquetar argumentos posicionales.

Si utilizamos el operador * delante del nombre de un parámetro posicional, estaremos indicando que los argumentos pasados a la función se empaqueten en una tupla:

```
>>> def test_args(*args):
...     print(f'{args=}')
...
>>> test_args()
args=()

>>> test_args(1, 2, 3, 'pescado', 'salado', 'es')
args=(1, 2, 3, 'pescado', 'salado', 'es')
```

Nota: El hecho de llamar args al parámetro es una convención.

También podemos utilizar esta estrategia para establecer en una función una serie de parámetros como requeridos y recibir el resto de argumentos como opcionales y empaquetados:

```
>>> def sum_all(v1, v2, *args):
...     total = 0
...     for value in (v1, v2) + args:
...         total += value
...     return total
...
>>> sum_all()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: sum_all() missing 2 required positional arguments: 'v1' and 'v2'

>>> sum_all(1, 2)
3

>>> sum_all(5, 9, 3, 8, 11, 21)
57
```

Existe la posibilidad de usar el asterisco * en la llamada a la función para desempaquetar los argumentos posicionales:

```
>>> def test_args(*args):
...     print(f'{args=}')
...
>>> my_args = (4, 3, 7, 9)

>>> test_args(my_args) # No existe desempaquetado!
args=((4, 3, 7, 9),)
```



```
>>> test_args(*my_args) # Sí existe desempaquetado!  
args=(4, 3, 7, 9)
```

Empaquetar/Desempaquetar argumentos nominales.

Si utilizamos el operador ****** delante del nombre de un parámetro nominal, estaremos indicando que los argumentos pasados a la función se empaqueten en un diccionario:

```
>>> def test_kwargs(**kwargs):  
...     print(f'{kwargs=}')  
...  
>>> test_kwargs()  
kwargs={}
```

```
>>> test_kwargs(a=4, b=3, c=7, d=9)  
kwargs={'a': 4, 'b': 3, 'c': 7, 'd': 9}
```

Nota: El hecho de llamar **kwargs** al parámetro es una convención.

Al igual que veíamos previamente, existe la posibilidad de usar doble asterisco ****** en la llamada a la función para desempaquetar los argumentos nominales:

```
>>> def test_kwargs(**kwargs):  
...     print(f'{kwargs=}')  
...  
>>> my_kwargs = {'a': 4, 'b': 3, 'c': 7, 'd': 9}  
>>> test_kwargs(my_kwargs) # No existe desempaquetado!  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: test_kwargs() takes 0 positional arguments but 1 was given
```

El error anterior se produce porque estamos pasando un parámetro (**my_kwargs**) como posicional, cuando lo está esperando nominal.

```
>>> test_kwargs(**my_kwargs) # Sí existe desempaquetado!  
kwargs={'a': 4, 'b': 3, 'c': 7, 'd': 9}
```

Ahora no hay error porque, aunque pasamos un parámetro posicional, al usar ****** lo desempaquetamos y lo transforma en varios parámetros nominales.

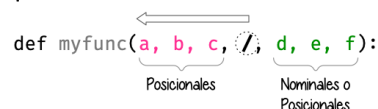
Forzando modo de paso de argumentos.

Si bien Python nos da flexibilidad para pasar argumentos a nuestras funciones en modo posicional o nominal, existen opciones para forzar a que dicho paso sea obligatorio en una determinada modalidad.

Argumentos sólo posicionales.

A partir de Python 3.8 se ofrece la posibilidad de obligar a que determinados parámetros de la función sean pasados sólo por posición. Para ello, en la definición de los parámetros de la función, tendremos que incluir un parámetro especial **/** que delimitará el tipo de parámetros. Así, todos los parámetros a la izquierda del delimitador estarán obligados a ser posicionales:

```
def myfunc(a, b, c, /, d, e, f):  
    ...  
    ...  
    ...
```



```
>>> def sum_power(a, b, /, power=False):
...     if power:
...         a **= 2
...         b **= 2
...     return a + b
...
>>> sum_power(3, 4)
7
>>> sum_power(3, 4, True)
25
>>> sum_power(3, 4, power=True)
25
>>> sum_power(a=3, b=4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: sum_power() got some positional-only arguments passed as keyword _
->arguments: 'a, b'
```

Argumentos sólo nominales.

A partir de Python 3 se ofrece la posibilidad de obligar a que determinados parámetros de la función sean pasados sólo por nombre. Para ello, en la definición de los parámetros de la función, tendremos que incluir un parámetro especial `*` que delimitará el tipo de parámetros. Así, todos los parámetros a la derecha del separador estarán obligados a ser nominales:



```
def myfunc(a, b, c, *, d, e, f):
```

Posicionales o Nominales Nominales

```
>>> def sum_power(a, b, *, power=False):
...     if power:
...         a **= 2
...         b **= 2
...     return a + b
...
>>> sum_power(3, 4)
7
>>> sum_power(a=3, b=4)
7
>>> sum_power(3, 4, power=True)
25
>>> sum_power(3, 4, True)
-----
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: sum_power() takes 2 positional arguments but 3 were given
```

Fijando argumentos posicionales y nominales.

Si mezclamos las dos estrategias anteriores podemos forzar a que una función reciba argumentos de un modo concreto. Continuando con ejemplo anterior, podríamos hacer lo siguiente:

```
>>> def sum_power(a, b, /, *, power=False):
...     if power:
```

```

...         a **= 2
...         b **= 2
...     return a + b
...
>>> sum_power(3, 4, power=True)    #Único modo posible de llamada

```

Funciones como parámetros.

Las funciones se pueden utilizar en cualquier contexto de nuestro programa. Son objetos que pueden ser asignados a variables, usados en expresiones, devueltos como valores de retorno o pasados como argumentos a otras funciones. Veamos un primer ejemplo en el que pasamos una función como argumento:

```

>>> def success():
...     print('Yeah!')
...
>>> type(success)
function

```

```

>>> def doit(f):
...     f()
...
>>> doit(success)
Yeah!

```

Veamos un segundo ejemplo en el que pasamos, no sólo una función como argumento, sino los valores con los que debe operar:

```

>>> def repeat_please(text, times=1):
...     return text * times
...
>>> type(repeat_please)
function

>>> def doit(f, arg1, arg2):
...     return f(arg1, arg2)
...
>>> doit(repeat_please, 'Funciones como parametros', 2)
'Funciones como parametrosFunciones como parametros'

```

6.1.3 Documentación.

Ya hemos visto que en Python podemos incluir comentarios para explicar mejor determinadas zonas de nuestro código. Del mismo modo podemos (y en muchos casos debemos) adjuntar documentación a la definición de una función incluyendo una cadena de texto (docstring) al comienzo de su cuerpo:

```

>>> def sqrt(value):
...     'Returns the square root of the value'
...     return value ** (1/2)
...

```

La forma más ortodoxa de escribir un docstring es utilizando triples comillas:

```

>>> def closest_int(value):
...     """Returns the closest integer to the given value.
...     The operation is:
...         1. Compute distance to floor.

```

```

...         2. If distance less than a half, return floor.
...         Otherwise, return ceil.
...     """
...     floor = int(value)
...     if value - floor < 0.5:
...         return floor
...     else:
...         return floor + 1
...

```

Para ver el docstring de una función, basta con utilizar help:

```

>>> help(closest_int)
Help on function closest_int in module __main__:
closest_int(value)
    Returns the closest integer to the given value.
    The operation is:
        1. Compute distance to floor.
        2. If distance less than a half, return floor.
        Otherwise, return ceil.

```

También es posible extraer información usando el símbolo de interrogación:

```

>>> closest_int?
Signature: closest_int(value)
Docstring:
Returns the closest integer to the given value.
The operation is:
    1. Compute distance to floor.
    2. If distance less than a half, return floor.
    Otherwise, return ceil.
File: ~/aprendepython/<ipython-input-75-5dc166360da1>
Type: function

```

Importante: Esto no sólo se aplica a funciones creadas por nosotros, sino a cualquier otra función definida en el lenguaje.

Explicación de parámetros.

Para documentar una función utilizando un docstring podemos utilizar diferentes formatos que se han estandarizado. Cada uno de ellos tiene una redacción y particularidades específicas, pero todos comparten una misma estructura:

- ✓ Una primera línea de descripción de la función.
- ✓ A continuación, especificamos las características de los parámetros (incluyendo sus tipos).
- ✓ Por último, indicamos si la función retorna un valor y sus características.

El siguiente ejemplo se centra en el formato “Sphinx docstrings”:

```

>>> def my_power(x, n):
...     """Calculate x raised to the power of n.
...
...     :param x: number representing the base of the operation
...     :type x: int
...     :param n: number representing the exponent of the operation
...     :type n: int

```

```

...
...     :return: :math:x^n
...     :rtype: int
...     """
...     result = 1
...     for _ in range(n):
...         result *= x
...     return result
...

```

6.1.4 Tipos de funciones.

Funciones interiores.

Está permitido definir una función dentro de otra función:

```

>>> def validation_test(text):
...     def is_valid_char(char):
...         return char in 'xyz'
...     checklist = []
...     for char in text:
...         checklist.append(is_valid_char(char))
...     return sum(checklist) / len(text)
...
>>> validation_test('xyzxyz')
1.0
>>> validation_test('abzxyabcdz')
0.4
>>> validation_test('abc')
0.0

```

Clausuras.

Una clausura (del término inglés «closure») establece el uso de una función interior que se genera dinámicamente y recuerda los valores de los argumentos con los que fue creada:

```

>>> def make_multiplier_of(n):
...     def multiplier(x):
...         return x * n
...     return multiplier
...
>>> m3 = make_multiplier_of(3)
>>> m5 = make_multiplier_of(5)
>>> type(m3)
function
>>> m3(7) # 7 * 3
21
>>> type(m5)
function
>>> m5(8) # 8 * 5
40

```

Importante: en una clausura retornamos una función, no una llamada a la función.

Funciones anónimas «lambda».

Una función lambda tiene las siguientes propiedades:

- ✓ Se escribe con una única sentencia.
- ✓ No tiene nombre (es anónima).
- ✓ Su cuerpo tiene implícito un return.
- ✓ Puede recibir cualquier número de parámetros.

Veamos un primer ejemplo de función «lambda» que nos permite contar el número de palabras de una cadena de texto:

```
>>> num_words = lambda t: len(t.strip().split())
>>> type(num_words)
function
>>> num_words
<function __main__.<lambda>(t)>
>>> num_words('hola amigo que tal estás')
5
```

Veamos otro ejemplo en el que mostramos una tabla con el resultado de aplicar el «and» lógico mediante una función «lambda» que ahora recibe dos parámetros:

```
>>> logic_and = lambda x, y: x & y
>>> for i in range(2):
...     for j in range(2):
...         print(f'{i} & {j} = {logic_and(i, j)}')
...
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

Las funciones «lambda» son bastante utilizadas como argumentos a otras funciones. Un ejemplo claro de ello es la función sorted que tiene un parámetro opcional key donde se define la clave de ordenación.

Veamos cómo usar una función anónima «lambda» para ordenar una tupla de pares longitud-latitud:

```
>>> geoloc = (
... (15.623037, 13.258358),
... (55.147488, -2.667338),
... (54.572062, -73.285171),
... (3.152857, 115.327724),
... (-40.454262, 172.318877)
)
>>> # Ordenación por longitud (primer elemento de la tupla)
>>> sorted(geoloc)
[(-40.454262, 172.318877),
(3.152857, 115.327724),
(15.623037, 13.258358),
(54.572062, -73.285171),
(55.147488, -2.667338)]
>>> # Ordenación por latitud (segundo elemento de la tupla)
>>> sorted(geoloc, key=lambda t: t[1])
[(54.572062, -73.285171),
```

```
(55.147488, -2.667338),  
(15.623037, 13.258358),  
(3.152857, 115.327724),  
(-40.454262, 172.318877)]
```

Decoradores.

Hay situaciones en las que necesitamos modificar el comportamiento de funciones existentes pero sin alterar su código. Para estos casos es muy útil usar decoradores.

Un decorador es una función que recibe como parámetro una función y devuelve otra función.

Se podría ver como un caso particular de clausura.

Veamos un ejemplo en el que indicamos que se está ejecutando una función:

```
>>> def simple_logger(func):  
...     def wrapper(*args, **kwargs):  
...         print(f'Running "{func.__name__}"...')  
...         return func(*args, **kwargs)  
...     return wrapper  
...  
>>> type(simple_logger)  
function
```

Ahora vamos a definir una función ordinaria (que usaremos más adelante):

```
>>> def hi(name):  
...     return f'Hello {name}!'  
...  
>>> hi('Guido')  
Hello Guido!  
>>> hi('Lovelace')  
Hello Lovelace!
```

Ahora aplicaremos el decorador definido previamente `simple_logger()` sobre la función ordinaria `hi()`. Se dice que `simple_logger()` es la función decoradora y que `hi()` es la función decorada. De esta forma obtendremos mensajes informativos adicionales.

Además, el decorador es aplicable a cualquier número y tipo de argumentos e incluso a cualquier otra función ordinaria:

```
>>> decorated_hi = simple_logger(hi)  
>>> decorated_hi('Guido')  
Running "hi"...  
'Hello Guido!'  
>>> decorated_hi('Lovelace')  
Running "hi"...  
'Hello Lovelace!'
```

Usando @ para decorar.

Python nos ofrece un «syntactic sugar» para simplificar la aplicación de los decoradores a través del operador `@` justo antes de la definición de la función que queremos decorar:

```
>>> @simple_logger  
... def hi(name):  
...     return f'Hello {name}!'  
...  
...
```

```
>>> hi('Galindo')
Running "hi"...
'Hello Galindo!'
>>> hi('Terrón')
Running "hi"...
'Hello Terrón!'
```

Podemos aplicar más de un decorador a cada función. Para ejemplificarlo vamos a crear dos decoradores muy sencillos:

```
>>> def plus5(func):
...     def wrapper(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result + 5
...     return wrapper
...
>>> def div2(func):
...     def wrapper(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result // 2
...     return wrapper
...
```

Ahora aplicaremos ambos decoradores sobre una función que realiza el producto de dos números:

```
>>> @plus5
... @div2
... def prod(a, b):
...     return a * b
...
>>> prod(4, 3)
11
>>> ((4 * 3) // 2) + 5    #esta es la operación que realiza en realidad prod(4, 3) con los decoradores
11
```

Importante: cuando tenemos varios decoradores aplicados a una función, el orden de ejecución empieza por aquel decorador más «cercano» a la definición de la función.

Funciones recursivas.

La recursividad es el mecanismo por el cual una función se llama a sí misma:

```
>>> def call_me():
...     return call_me()
...
```

```
>>> call_me()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in call_me
File "<stdin>", line 2, in call_me
File "<stdin>", line 2, in call_me
[Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Advertencia: podemos observar que existe un número máximo de llamadas recursivas.

Python controla esta situación por nosotros, ya que, de no ser así, podríamos llegar a consumir los recursos del sistema.

Veamos ahora un ejemplo más real en el que computar el enésimo término de la Sucesión de Fibonacci utilizando una función recursiva:

```
>>> def fibonacci(n):
...     if n == 0:
...         return 0
...     if n == 1:
...         return 1
...     return fibonacci(n - 1) + fibonacci(n - 2)
...
>>> fibonacci(10)
55
>>> fibonacci(20)
6765
```

6.1.5 Espacios de nombres.

Los espacios de nombres permiten definir ámbitos o contextos en los que agrupar nombres de objetos. Proporcionan un mecanismo de empaquetamiento, de tal forma que podamos tener incluso nombres iguales que no hacen referencia al mismo objeto (siempre y cuando estén en ámbitos distintos).

Cada función define su propio espacio de nombres y es diferente del espacio de nombres global aplicable a todo nuestro programa.

Acceso a variables globales.

Cuando una variable se define en el espacio de nombres global podemos hacer uso de ella con total transparencia dentro del ámbito de las funciones del programa:

```
>>> language = 'castellano'
>>> def catalonia():
...     print(f'{language=}')
...
>>> language
'castellano'
>>> catalonia()
language='castellano'
```

Creando variables locales.

En el caso de que asignemos un valor a una variable global dentro de una función, no estaremos modificando ese valor, sino que estaremos creando una variable en el espacio de nombres local:

```
>>> language = 'castellano'
>>> def catalonia():
...     language = 'catalan'
...     print(f'{language=}')
...
>>> language
'castellano'
>>> catalonia()
language='catalan'
>>> language
'castellano'
```

Forzando modificación global.

Python nos permite modificar una variable definida en un espacio de nombres global dentro de una función. Para ello debemos usar el modificador “global”:

```
>>> language = 'castellano'
>>> def catalonia():
...     global language
...     language = 'catalan'
...     print(f'{language=}')
...
>>> language
'castellano'
>>> catalonia()
language='catalan'
>>> language
'catalan'
```

Advertencia: el uso de global no se considera una buena práctica ya que puede inducir a confusión y tener efectos colaterales indeseados.

Contenido de los espacios de nombres.

Python proporciona dos funciones para acceder al contenido de los espacios de nombres:

- ✓ `locals()`: devuelve un diccionario con los contenidos del espacio de nombres local.
- ✓ `globals()`: devuelve un diccionario con los contenidos del espacio de nombres global.

```
>>> language = 'castellano'
>>> def catalonia():
...     language = 'catalan'
...     print(f'{locals()=}')
...
>>> language
'castellano'
>>> catalonia()
locals()={'language': 'catalan'}
```

```
>>> globals()
```

#no reproduciremos el contenido del diccionario del espacio de nombres global por ser muy extenso

6.2 Objetos y Clases.

Hasta ahora hemos estado usando objetos de forma totalmente transparente, casi sin ser conscientes de ello. Pero, en realidad, todo en Python es un objeto, desde números a funciones. Esto se debe a que el lenguaje provee ciertos mecanismos para no tener que usar explícitamente técnicas de orientación a objetos.

Llegados a este punto, investigaremos en profundidad sobre la creación y manipulación de clases y objetos, y todas las operaciones que engloban este paradigma.

6.2.1 Programación orientada a objetos

La programación orientada a objetos (POO) es una manera de programar que permite llevar al código mecanismos usados con entidades de la vida real.

Sus beneficios son los siguientes:

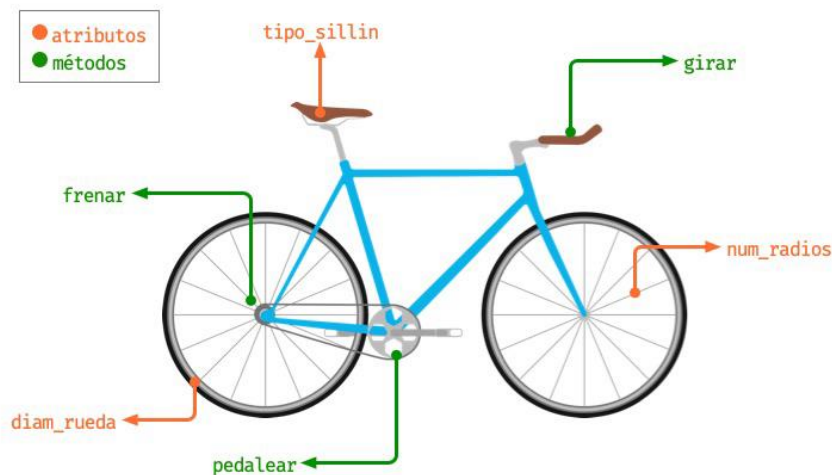
- ✓ Encapsulamiento: permite **empaquetar** el código dentro de una unidad (objeto) donde se puede determinar el ámbito de actuación.
- ✓ Abstracción: permite **generalizar** los tipos de objetos a través de las clases y simplificar el programa.
- ✓ Herencia: permite **reutilizar** código al poder heredar atributos y comportamientos de una clase a otra.
- ✓ Polimorfismo: permite **crear** múltiples objetos a partir de una misma pieza flexible de código.

¿Qué es un objeto?

Un objeto es una estructura de datos personalizada que contiene datos y código:

Elementos	¿Qué son?	¿Cómo se llaman?	¿Cómo se identifican?
Datos	Variables	Atributos	Nombres
Código	Funciones	Métodos	Verbos

Un objeto representa una instancia única de alguna entidad a través de los valores de sus atributos e interactúan con otros objetos (o consigo mismo) a través de sus métodos.



¿Qué es una clase?

Para crear un objeto primero debemos definir la clase que lo contiene. Podemos pensar en la clase como el molde con el que crear nuevos objetos de ese tipo.

En el proceso de diseño de una clase hay que tener en cuenta (entre otros) el principio de responsabilidad única, intentando que los atributos y los métodos que contenga estén enfocados a un objetivo único y bien definido.

6.2.2 Creando objetos.

Empecemos por crear nuestra primera clase. En este caso vamos a modelar algunos de los droides de la saga StarWars:

Para ello usaremos la palabra reservada `class` seguido del nombre de la clase:

```
>>> class StarWarsDroid:  
...     pass  
...
```

Consejo: los nombres de clases se suelen escribir en formato CamelCase y en singular.

Existen multitud de droides en el universo StarWars. Una vez que hemos definido la clase genérica podemos crear instancias/objetos (droides) concretos:

```
>>> c3po = StarWarsDroid()
>>> r2d2 = StarWarsDroid()
>>> bb8 = StarWarsDroid()
>>> type(c3po)
__main__.StarWarsDroid
>>> type(r2d2)
__main__.StarWarsDroid
>>> type(bb8)
__main__.StarWarsDroid
```

Añadiendo atributos.

Un atributo no es más que una variable, un nombre al que asignamos un valor, con la particularidad de vivir dentro de una clase o de un objeto.

Los atributos, por lo general, se suelen asignar durante la creación de un objeto, pero también es posible añadirlos a posteriori:

```
>>> blue_droid = StarWarsDroid()
>>> golden_droid = StarWarsDroid()
>>> golden_droid.name = 'C-3PO'
>>> blue_droid.name = 'R2-D2'
>>> blue_droid.height = 1.09
>>> blue_droid.num_feet = 3
>>> blue_droid.partner_droid = golden_droid # otro droide como atributo
```

Una vez creados, es muy sencillo acceder a los atributos:

```
>>> golden_droid.name
'C-3PO'
>>> blue_droid.num_feet
3
```

Hemos definido un droide «socio». Veremos a continuación que podemos trabajar con él de una manera totalmente natural:

```
>>> type(blue_droid.partner_droid)
__main__.StarWarsDroid
>>> blue_droid.partner_droid.name # acceso al nombre del droide socio
'C-3PO'
>>> blue_droid.partner_droid.num_feet # aún sin definir!
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'StarWarsDroid' object has no attribute 'num_feet'
>>> blue_droid.partner_droid.num_feet = 2
```

Añadiendo métodos.

Un método es una función que forma parte de una clase o de un objeto. En su ámbito tiene acceso a otros métodos y atributos de la clase o del objeto al que pertenece.

La definición de un método (de instancia) es análoga a la de una función ordinaria, pero incorporando un primer parámetro self que hace referencia a la instancia actual del objeto.

Una de las acciones más sencillas que se pueden hacer sobre un droide es encenderlo o apagarlo. Vamos a implementar estos dos métodos en nuestra clase:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...
>>> k2so = Droid()
>>> k2so.switch_on()
Hi! I'm a droid. Can I help you?
>>> k2so.switch_off()
Bye! I'm going to sleep
```

Inicialización.

Existe un método especial que se ejecuta cuando creamos una instancia de un objeto. Este método es `__init__` y nos permite asignar atributos y realizar operaciones con el objeto en el momento de su creación. También es ampliamente conocido como el “constructor”.

Veamos un ejemplo de este método con nuestros droides en el que únicamente guardaremos el nombre del droide como un atributo del objeto:

```
1 >>> class Droid:
2 ...     def __init__(self, name):
3 ...         self.name = name
4 ...
5
6 >>> droid = Droid('BB-8')
7
8 >>> droid.name
9 'BB-8'
```

Aclaraciones:

- ✓ Línea 2: definición del constructor.
- ✓ Línea 7: creación del objeto (y llamada implícita al constructor).
- ✓ Línea 9: acceso al atributo name creado previamente en el constructor.

6.2.3 Atributos.

Acceso directo.

En el siguiente ejemplo vemos que, aunque el atributo name se ha creado en el constructor de la clase, también podemos modificarlo desde «fuera» con un acceso directo:

```
>>> class Droid:
...     def __init__(self, name):
...         self.name = name
...
>>> droid = Droid('C-3PO')
>>> droid.name
'C-3PO'
>>> droid.name = 'waka-waka' # esto sería válido!
```

Propiedades.

Como hemos visto previamente, los atributos definidos en un objeto son accesibles públicamente. Esto puede parecer extraño a personas desarrolladoras de otros lenguajes. En Python existe un cierto «sentido de responsabilidad» a la hora de programar y manejar este tipo de situaciones.

Una posible solución «pitónica» para la privacidad de los atributos es el uso de propiedades. La forma más común de aplicar propiedades es mediante el uso de decoradores:

- ✓ @property: para leer el valor de un atributo.
- ✓ @name.setter: para escribir el valor de un atributo.

Veamos un ejemplo en el que estamos ofuscando el nombre del droide a través de propiedades:

```
>>> class Droid:
...     def __init__(self, name):
...         self.hidden_name = name
...
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.hidden_name
...
...     @name.setter
...     def name(self, name):
...         print('inside the setter')
...         self.hidden_name = name
...
>>> droid = Droid('N1-G3L')
>>> droid.name
inside the getter
'N1-G3L'
>>> droid.name = 'Nigel'
inside the setter
>>> droid.name
inside the getter
'Nigel'
```

En cualquier caso, seguimos pudiendo acceder directamente a “.hidden_name”:

```
>>> droid.hidden_name
'Nigel'
```

Valores calculados.

Una propiedad también se puede usar para devolver un valor calculado (o computado). A modo de ejemplo, supongamos que la altura del periscopio de los droides astromecánicos se calcula siempre como un porcentaje de su altura. Veamos cómo implementarlo:

```
>>> class AstromechDroid:
...     def __init__(self, name, height):
...         self.name = name
...         self.height = height
...
...     @property
...     def periscope_height(self):
...         return 0.3 * self.height
...
```

```
>>> droid = AstromechDroid('R2-D2', 1.05)
>>> droid.periscope_height # podemos acceder como atributo
0.315
>>> droid.periscope_height = 10 # no podemos modificarlo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Consejo: la ventaja de usar valores calculados sobre simples atributos es que el cambio de valor en un atributo no asegura que actualicemos otro atributo, y además, siempre podremos modificar directamente el valor del atributo, con lo que podríamos obtener efectos colaterales indeseados.

Ocultando atributos.

Python tiene una convención sobre aquellos atributos que queremos hacer «privados» (u ocultos), comenzar el nombre con doble subrayón `__`

```
>>> class Droid:
...     def __init__(self, name):
...         self.__name = name
...
>>> droid = Droid('BC-44')
>>> droid.__name # efectivamente no aparece como atributo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Droid' object has no attribute '__name'
```

Lo que realmente ocurre tras el telón se conoce como «name mangling» y consiste en modificar el nombre del atributo incorporando la clase como un prefijo. Sabiendo esto podemos acceder al valor del atributo supuestamente privado:

```
>>> droid._Droid__name
'BC-44'
```

Atributos de clase.

Podemos asignar atributos a las clases y serán heredados por todos los objetos instanciados de esa clase. A modo de ejemplo, supondremos que, en un principio, todos los droides están diseñados para que obedezcan a su dueño. Esto lo conseguiremos a nivel de clase, salvo que ese comportamiento se sobrescriba en un objeto de esa clase :

```
>>> class Droid:
...     obeys_owner = True # obedece a su dueño
...
>>> good_droid = Droid()
>>> good_droid.obeyes_owner
True
>>> t1000 = Droid()
>>> t1000.obeyes_owner = False # T-1000 (Terminator)
>>> t1000.obeyes_owner
False
>>> Droid.obeyes_owner # el cambio no afecta a nivel de clase (sólo al objeto)
True
```

6.2.4 Métodos.

Métodos de instancia.

Un método de instancia es un método que modifica el comportamiento del objeto al que hace referencia. Recibe self como primer parámetro, el cual se convierte en el propio objeto sobre el que estamos trabajando. Python envía este argumento de forma transparente.

Veamos un ejemplo en el que, además del constructor, creamos un método de instancia para desplazar un droid:

```
>>> class Droid:
...     def __init__(self, name): # método de instancia -> constructor
...         self.name = name
...         self.covered_distance = 0
...
...     def move_up(self, steps): # método de instancia
...         self.covered_distance += steps
...         print(f'Moving {steps} steps')
...
>>> droid = Droid('C1-10P')
>>> droid.move_up(10)
Moving 10 steps
```

Métodos de clase.

Un método de clase es un método que modifica el comportamiento de la clase a la que hace referencia. Recibe cls como primer parámetro, el cual se convierte en la propia clase sobre la que estamos trabajando. Python envía este argumento de forma transparente. La identificación de estos métodos se completa aplicando el decorador @classmethod a la función.

Veamos un ejemplo en el que implementaremos un método de clase que lleva la cuenta de los droides que hemos creado:

```
>>> class Droid:
...     count = 0
...
...     def __init__(self):
...         Droid.count += 1
...
...     @classmethod
...     def total_droids(cls):
...         print(f'{cls.count} droids built so far!')
...
>>> droid1 = Droid()
>>> droid2 = Droid()
>>> droid3 = Droid()
>>> Droid.total_droids()
3 droids built so far!
```

Métodos estáticos.

Un método estático es un método que no modifica el comportamiento del objeto ni de la clase. No recibe ningún parámetro especial. La identificación de estos métodos se completa aplicando el decorador @staticmethod a la función.

Veamos un ejemplo en el que creamos un método estático para devolver las categorías de droides que existen en StarWars:

```
>>> class Droid:
...     def __init__(self):
...         pass
...
...     @staticmethod
...     def get_droids_categories():
...         return ['Messeger', 'Astromech', 'Power', 'Protocol']
...
>>> Droid.get_droids_categories()
['Messeger', 'Astromech', 'Power', 'Protocol']
```

Métodos mágicos.

Cuando escribimos 'hello world' * 3 ¿cómo sabe el objeto 'hello world' lo que debe hacer para multiplicarse con el objeto entero 3? O, dicho de otra forma, ¿cuál es la implementación interna del operador * para «strings» y enteros? En valores numéricos puede parecer evidente (siguiendo los operadores matemáticos), pero no es así para otros objetos. La solución que proporciona Python para estas (y otras) situaciones son los métodos mágicos.

Por tanto, los métodos mágicos son implementaciones internas de ciertos operadores (o estructuras o expresiones), de manera que, cuando nosotros usamos alguno de estos operadores, se dispara automáticamente el método mágico que lo implementa. Los métodos mágicos ya están creados, pero también los podemos crear (personalizar) nosotros en la clase que queramos para definir como actuará ese operador (método mágico) con esa clase.

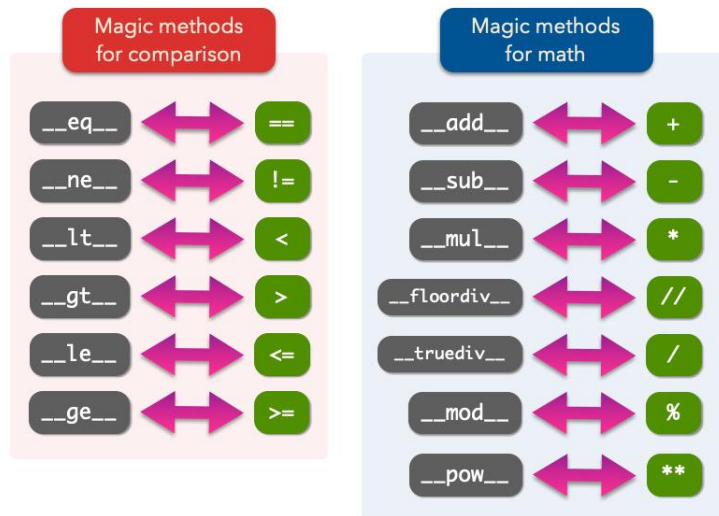
El nombre de los métodos mágicos empieza y termina por doble subguión “__”. Uno de los métodos mágicos más famosos es el constructor de una clase: __init__().

Vamos a ver un ejemplo con el operador de comparación aplicado a dos objetos, que se realiza con el método mágico __eq__(). Extrapolando esta idea a nuestro universo StarWars, podríamos establecer que dos droides son iguales si su nombre es igual, independientemente de que tengan distintos números de serie:

```
>>> class Droid:
...     def __init__(self, name, serial_number):
...         self.serial_number = serial_number
...         self.name = name
...
...     def __eq__(self, droid):
...         return self.name == droid.name
...
>>> droid1 = Droid('C-3PO', 43974973242)
>>> droid2 = Droid('C-3PO', 85094905984)
>>> droid1 == droid2    # llamada implícita a __eq__
True
>>> droid1.__eq__(droid2)    # llamada explícita a __eq__
True
```

Nota: Los métodos mágicos no sólo están restringidos a operadores de comparación o matemáticos. Existen muchos otros en la documentación oficial de Python, donde son llamados “métodos especiales”.

No obstante, en la siguiente imagen se muestran los principales métodos mágicos para los operadores de comparación y operadores matemáticos:



Veamos otro ejemplo en el que «sumamos» dos droides. Esto se podría ver como una fusión. Supongamos que la suma de dos droides implica:

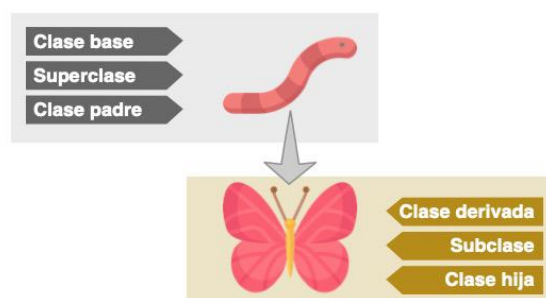
- ✓ Que el nombre del droide resultante es la concatenación de los nombres de los droides.
- ✓ Que la energía del droide resultante es la suma de la energía de los droides:

```
>>> class Droid:
...     def __init__(self, name, power):
...         self.name = name
...         self.power = power
...
...     def __add__(self, droid):
...         new_name = self.name + '-' + droid.name
...         new_power = self.power + droid.power
...         return Droid(new_name, new_power) # Hay que devolver un objeto de tipo Droid
...
>>> droid1 = Droid('C3PO', 45)
>>> droid2 = Droid('R2D2', 91)
>>> droid3 = droid1 + droid2
>>> print(f'Fusion droid:\n{droid3.name} with power {droid3.power}')
Fusion droid:
C3PO-R2D2 with power 136
```

6.2.5 Herencia.

La herencia consiste en crear una nueva clase partiendo de una clase existente, pero que añade o modifica ciertos aspectos. Se considera una buena práctica tanto para reutilizar código como para realizar generalizaciones.

Nota: Cuando se utiliza herencia, la clase derivada, de forma automática, puede usar todo el código de la clase base sin necesidad de copiar nada explícitamente.



Heredar desde una clase base.

Para que una clase «herede» de otra, basta con indicar la clase base entre paréntesis en la definición de la clase derivada.

Sigamos con el ejemplo. Una de las grandes categorías de droides en StarWars es la de droides de protocolo. Vamos a crear una herencia sobre esta idea:

```
>>> class Droid:
...     """ Clase Base """
...     pass
...
>>> class ProtocolDroid(Droid):
...     """ Clase Derivada """
...     pass
...
>>> issubclass(ProtocolDroid, Droid) # comprobación de herencia
True
>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()
```

Vamos a añadir un par de métodos a la clase base, y analizar su comportamiento:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...
>>> class ProtocolDroid(Droid):
...     pass
...
>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()
>>> r2d2.switch_on()
Hi! I'm a droid. Can I help you?
>>> c3po.switch_on() # método heredado de Droid
Hi! I'm a droid. Can I help you?
>>> r2d2.switch_off()
Bye! I'm going to sleep
```

Sobreescribir un método.

Como hemos visto, una clase derivada hereda todo lo que tiene su clase base. Pero en muchas ocasiones nos interesa modificar el comportamiento de esta herencia. En el ejemplo vamos a modificar el comportamiento del método `switch_on()` para la clase derivada:

```
>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...
>>> class ProtocolDroid(Droid):
...     def switch_on(self):
```

```

...         print("Hi! I'm a PROTOCOL droid. Can I help you?")
...
>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()
>>> r2d2.switch_on()
Hi! I'm a droid. Can I help you?
>>> c3po.switch_on() # método heredado pero sobreescrito
Hi! I'm a PROTOCOL droid. Can I help you?

```

Añadir un método.

La clase derivada también puede añadir métodos que no estaban presentes en su clase base. En el siguiente ejemplo vamos a añadir un método `translate()` que permita a los droides de protocolo traducir cualquier mensaje:

```

>>> class Droid:
...     def switch_on(self):
...         print("Hi! I'm a droid. Can I help you?")
...
...     def switch_off(self):
...         print("Bye! I'm going to sleep")
...
>>> class ProtocolDroid(Droid):
...     def switch_on(self):
...         print("Hi! I'm a PROTOCOL droid. Can I help you?")
...
...     def translate(self, msg, from_language):
...         """ Translate from language to Human understanding """
...         print(f'{msg} means "ZASCA" in {from_language}')
>>> r2d2 = Droid()
>>> c3po = ProtocolDroid()
>>> c3po.translate('kiitos', 'Huttese') # idioma de Watoo
kiitos means "ZASCA" in Huttese
>>> r2d2.translate('kiitos', 'Huttese') # droide genérico no puede traducir
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'Droid' object has no attribute 'translate'

```

Con esto ya hemos aportado una personalidad diferente a los droides de protocolo, a pesar de que heredan de la clase genérica de droides de StarWars.

Accediendo a la clase base.

Puede darse la situación en la que tengamos que acceder desde la clase derivada a métodos o atributos de la clase base. Python ofrece "`super()`" como mecanismo para ello. Veamos un ejemplo más elaborado con nuestros droides:

```

>>> class Droid:
...     def __init__(self, name):
...         self.name = name
...
>>> class ProtocolDroid(Droid):
...     def __init__(self, name, languages):
...         super().__init__(name) # llamada al constructor de la clase base
...         self.languages = languages
...

```

```
>>> droid = ProtocolDroid('C-3PO', ['Ewokese', 'Huttese', 'Jawaese'])
>>> droid.name # fijado en el constructor de la clase base
'C-3PO'
>>> droid.languages # fijado en el constructor de la clase derivada
['Ewokese', 'Huttese', 'Jawaese']
```

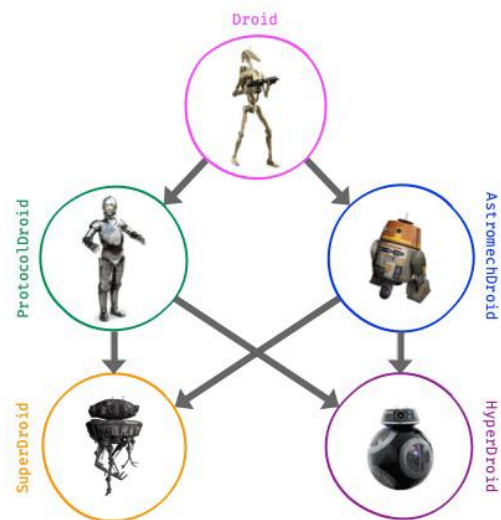
Herencia múltiple.

Aunque no está disponible en todos los lenguajes de programación, Python sí permite que los objetos puedan heredar de múltiples clases base.

Si en una clase se hace referencia a un método o atributo que no existe, Python lo buscará en todas sus clases base. Es posible que exista una colisión en caso de que el método o el atributo buscado esté, a la vez, en varias clases base. En este caso, Python resuelve el conflicto a través del orden de resolución de métodos (en inglés “mro”).

Supongamos que queremos modelar la siguiente estructura de clases con herencia múltiple:

```
>>> class Droid:
...     def greet(self):
...         return 'Here a droid'
...
>>> class ProtocolDroid(Droid):
...     def greet(self):
...         return 'Here a protocol droid'
...
>>> class AstromechDroid(Droid):
...     def greet(self):
...         return 'Here an astromech droid'
...
>>> class SuperDroid(ProtocolDroid, AstromechDroid):
...     pass
...
>>> class HyperDroid(AstromechDroid, ProtocolDroid):
...     pass
```



Todas las clases en Python disponen de un método especial llamado `mro()` que devuelve una lista de las clases que se visitarían (y en qué orden) en caso de acceder a un método o un atributo. También existe el atributo `__mro__` que es una tupla de esas clases:

```
>>> SuperDroid.mro()
[__main__.SuperDroid,
 __main__.ProtocolDroid,
 __main__.AstromechDroid,
 __main__.Droid,
 object]
>>> HyperDroid.__mro__
(__main__.HyperDroid,
 __main__.AstromechDroid,
 __main__.ProtocolDroid,
 __main__.Droid,
 object)
```

Veamos el resultado de la llamada a los métodos definidos:

```
>>> super_droid = SuperDroid()
>>> hyper_droid = HyperDroid()
>>> super_droid.greet()      #se encuentra primero con ProtocolDroid
'Here a protocol droid'
>>> hyper_droid.greet()     #se encuentra primero con AstromechDroid
'Here an astromech droid'
```

Nota: Todos los objetos en Python heredan, en primera instancia, de object. Esto se puede comprobar con el mro() correspondiente:

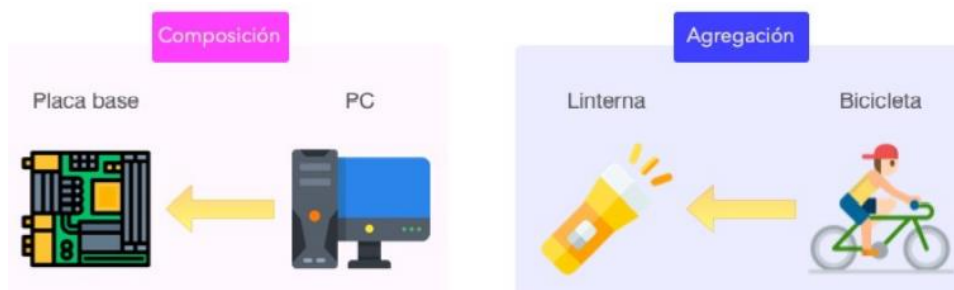
```
>>> int.mro()
[int, object]
>>> str.mro()
[str, object]
>>> list.mro()
[list, object]
```

Agregación y composición.

Aunque la herencia de clases nos permite modelar una gran cantidad de casos de uso en términos de «is-a» (es un), existen muchas otras situaciones en las que la agregación o la composición son una mejor opción. En este caso una clase se compone de otras clases, por tanto, hablamos de relaciones «has-a» (tiene un).

Hay una sutil diferencia entre agregación y composición:

- ✓ La composición implica que el objeto utilizado no puede «funcionar» sin la presencia de su propietario.
- ✓ La agregación implica que el objeto utilizado puede funcionar por sí mismo.



Veamos un ejemplo de agregación en el que añadimos una herramienta a un droide:

```
>>> class Tool:
...     def __init__(self, name):
...         self.name = name
...
...     def __str__(self):
...         return self.name.upper()
...
... class Droid:
...     def __init__(self, name, serial_number, tool):
...         self.name = name
...         self.serial_number = serial_number
...         self.tool = tool # agregación
...
```

```

...     def __str__(self):
...         return f'Droid {self.name} armed with a {self.tool}'
...
>>> lighter = Tool('lighter')
>>> bb8 = Droid('BB-8', 48050989085439, lighter)
>>> print(bb8)
Droid BB-8 armed with a LIGHTER

```

6.3 Módulos.

Escribir pequeños trozos de código puede resultar interesante para realizar determinadas pruebas. Pero a la larga, nuestros programas tenderán a crecer y será necesario agrupar el código en unidades manejables.

Los módulos son simplemente ficheros de texto que contienen código Python y representan unidades con las que evitar la repetición y favorecer la reutilización.

6.3.1 Importar un módulo.

Para hacer uso del código de otros módulos usaremos la sentencia "import". Esto permite importar el código y las variables de dicho módulo para que estén disponibles en nuestro programa.

La forma más sencilla de importar un módulo es "import <module>" donde module es el nombre de otro fichero Python, sin la extensión .py. Supongamos que partimos del siguiente fichero (módulo):

arith.py

```

1 def addere(a, b):
2     """Sum of input values"""
3     return a + b
4
5
6 def minuas(a, b):
7     """Substract of input values'
8     return a - b
9
10
11 def pullulate(a, b):
12     """Product of input values"""
13     return a * b
14
15
16 def partitus(a, b):
17     """Division of input values"""
18     return a / b

```

Desde otro fichero (en principio en la misma carpeta) podríamos hacer uso de las funciones definidas en "arith.py".

Importar módulo completo.

Desde otro fichero haríamos lo siguiente para importar todo el contenido del módulo arith.py:

```
1 >>> import arith
2
3 >>> arith.addere(3, 7)
4 10
```

Nota: en la línea 3 debemos anteponer, a la función addere(), el espacio de nombres que define el módulo arith.

Ruta de búsqueda de módulos.

Python tiene 2 formas de encontrar un módulo:

- ✓ En la carpeta actual de trabajo.
- ✓ En las rutas definidas en la variable de entorno PYTHONPATH.

Para ver las rutas de búsqueda establecidas, podemos ejecutar lo siguiente en un intérprete de Python:

```
>>> import sys
>>> sys.path
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',
'/path/to/.pyenv/versions/3.9.1/lib/python3.9',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',
'']
```

La cadena vacía que existe al final de la lista hace referencia a la carpeta actual.

Modificando la ruta de búsqueda.

Si queremos modificar la ruta de búsqueda, existen dos opciones:

Modificando directamente la variable PYTHONPATH.

Para ello exportamos dicha variable de entorno desde una terminal:

```
$ export PYTHONPATH=/tmp
```

Y comprobamos que se ha modificado en sys.path:

```
>>> sys.path
>>> sys.path
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',
'tmp',
'/path/to/.pyenv/versions/3.9.1/lib/python3.9',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',
'']
```

Modificando directamente la lista sys.path.

Para ello accedemos a la lista que está en el módulo sys de la librería estándar:

```
>>> sys.path.append('/tmp') # añadimos al final
>>> sys.path
>>> sys.path
['/path/to/.pyenv/versions/3.9.1/envs/aprendepython/bin',
'/path/to/.pyenv/versions/3.9.1/lib/python3.9',
'/path/to/.pyenv/versions/3.9.1/envs/aprendepython/lib/python3.9/site-packages',
'',
'tmp']
```


Truco: el hecho de poner nuestra ruta al principio o al final de sys.path influye en la búsqueda, ya que si existen dos (o más módulos) que se llaman igual en nuestra ruta de búsqueda, Python usará el primero que encuentre.

Importar partes de un módulo.

Es posible que no necesitemos todo aquello que está definido en arith.py. Supongamos que sólo vamos a realizar divisiones. Para ello haremos lo siguiente:

```
1 >>> from arith import partitus
2
3 >>> partitus(5, 2)
4 2.5
```

Nota: en la línea 3 ya podemos hacer uso directamente de la función partitus() porque la hemos importado directamente. Este esquema tiene el inconveniente de la posible colisión de nombres, en aquellos casos en los que tuviéramos algún objeto con el mismo nombre que el objeto que estamos importando.

Importar usando un alias.

Hay ocasiones en las que interesa, por colisión de otros nombres o por mejorar la legibilidad, usar un nombre diferente del módulo (u objeto) que estamos importando. Python nos ofrece esta posibilidad a través de la sentencia “as”.

Supongamos que queremos importar la función del ejemplo anterior, pero con otro nombre:

```
>>> from arith import partitus as mydivision
>>> mydivision(5, 2)
2.5
```

6.3.2 Paquetes.

Un paquete es simplemente una carpeta que contiene ficheros .py. Además, permite tener una jerarquía con más de un nivel de subcarpetas anidadas.

Para ejemplificar este modelo vamos a crear un paquete llamado mymath que contendrá 2 módulos:

- ✓ arith.py: para operaciones aritméticas (ya visto anteriormente).
- ✓ logic.py: para operaciones lógicas.

El código del módulo de operaciones lógicas es el siguiente:

logic.py

```
1 def et(a, b):
2     """Logic "and" of input values"""
3     return a & b
4
5
6 def uel(a, b):
7     """Logic "or" of input values"""
8     return a | b
9
10
11 def vel(a, b):
12     """Logic "xor" of input values"""
13     return a ^ b
```

Si nuestro código principal va a estar en un fichero main.py (a primer nivel), la estructura de ficheros nos quedaría de esta forma:

```
1 .
2 |— main.py
3 |— mymath
4     |— arith.py
5     |— logic.py
6
7 1 directory, 3 files
```

Aclaraciones:

- ✓ Línea 2: punto de entrada de nuestro programa a partir del fichero main.py.
- ✓ Línea 3: carpeta que define el paquete mymath.
- ✓ Línea 4: módulo para operaciones aritméticas.
- ✓ Línea 5: módulo para operaciones lógicas.

Importar desde un paquete.

Si ya estamos en el fichero main.py (o a ese nivel) podremos hacer uso de nuestro paquete de la siguiente forma:

```
1 >>> from mymath import arith, logic
2
3 >>> arith.pullulate(4, 7)
4 28
5
6 >>> logic.et(1, 0)
7 0
```

Aclaraciones:

- ✓ Línea 1: importar los módulos arith y logic del paquete mymath.
- ✓ Línea 3: uso de la función pullulate que está definida en el módulo arith.
- ✓ Línea 5: uso de la función et que está definida en el módulo logic.

6.3.3 Programa principal.

Cuando decidimos desarrollar una pieza de software en Python, normalmente usamos distintos ficheros para ello. Algunos de esos ficheros se convertirán en módulos, otros se englobarán en paquetes y existirá uno en concreto que será nuestro punto de entrada, también llamado programa principal.

Consejo: suele ser una buena práctica llamar main.py al fichero que contiene nuestro programa principal.

La estructura que suele tener este programa principal es la siguiente:

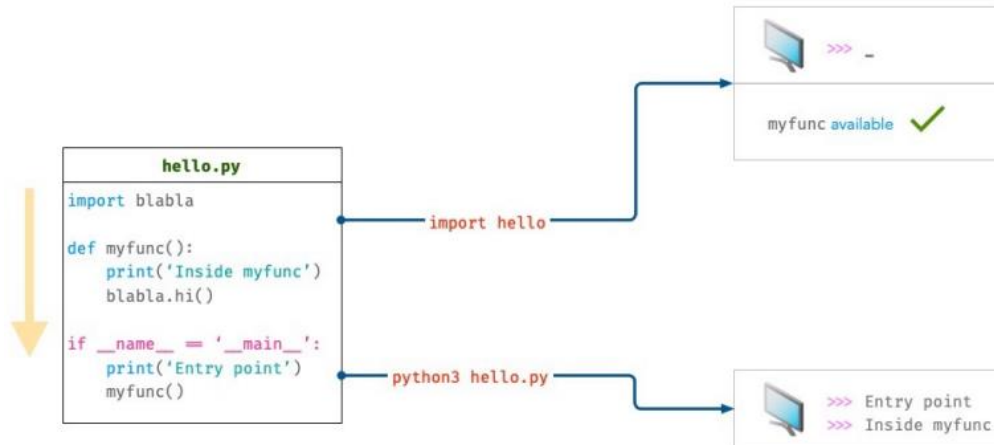
```
# imports de la librería estándar
# imports de librerías de terceros
# imports de módulos propios
# CÓDIGO PROPIO
# ...
# CÓDIGO PROPIO
if __name__ == '__main__':
    # punto de entrada real
```

Importante: si queremos ejecutar este fichero main.py desde línea de comandos, tendríamos que hacer:

\$ python3 main.py

if __name__ == __main__

Esta condición permite, en el programa principal, diferenciar qué código se lanzará cuando el fichero se ejecuta directamente o cuando el fichero se importa desde otro lugar.



En el ejemplo de la imagen, este sería el código del módulo:

hello.py

```
1 import blabla
2
3
4 def myfunc():
5     print('Inside myfunc')
6     blabla.hi()
7
8
9 if __name__ == '__main__':
10     print('Entry point')
11     myfunc()
```

Si se importa el módulo (import hello), el código se ejecuta de la siguiente forma:

- ✓ Línea 1: se importa el módulo blabla.
- ✓ Línea 4: se define la función myfunc() y estará disponible para usarse.
- ✓ Línea 9: esta condición no se cumple, ya que estamos importando y la variable especial __name__ no toma ese valor. De esta manera, finaliza la ejecución y no hay salida por pantalla.

Si se ejecuta directamente el módulo (\$ python3 hello.py), el código se ejecuta de la siguiente forma:

- ✓ Línea 1: se importa el módulo blabla.
- ✓ Línea 4: se define la función myfunc() y estará disponible para usarse.
- ✓ Línea 9: esta condición sí se cumple, ya que estamos ejecutando directamente el fichero (como programa principal) y la variable especial __name__ toma el valor __main__.
- ✓ Línea 10: salida por pantalla de la cadena de texto "Entry point".
- ✓ Línea 11: llamada a la función myfunc() que muestra por pantalla "Inside myfunc", además de invocar a la función hi() del módulo "blabla".

7. Excepciones.

Una excepción es el bloque de código que se lanza cuando se produce un error en la ejecución de un programa Python.

De hecho, ya hemos visto algunas de estas excepciones: accesos fuera de rango a listas o tuplas, accesos a claves inexistentes en diccionarios, etc. Cuando ejecutamos código que podría fallar bajo ciertas circunstancias, es necesario manejar de manera adecuada todas estas excepciones que se generan.

7.1. Manejando errores.

Si una excepción ocurre en algún lugar de nuestro programa y no es capturada en ese punto, va subiendo (burbujeando) hasta que es capturada en alguna función que ha hecho la llamada. Si en toda la «pila» de llamadas no existe un control de la excepción, Python muestra un mensaje de error con información adicional:

```
>>> def intdiv(a, b):
...     return a // b
...
>>> intdiv(3, 0)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in intdiv
ZeroDivisionError: integer division or modulo by zero
```

Para manejar (capturar) las excepciones podemos usar un bloque de código con las palabras reservadas “try” y “except”:

```
>>> def intdiv(a, b):
...     try:
...         return a // b
...     except:
...         print('Please do not divide by zero...')
...
>>> intdiv(3, 0)
Please do not divide by zero...
```

Aquel código que se encuentre dentro del bloque try se ejecutará normalmente siempre y cuando no haya un error. Si se produce una excepción, ésta será capturada por el bloque except, ejecutándose el código que contiene.

Consejo: no es una buena práctica usar un bloque except sin indicar el tipo de excepción que estamos gestionando, porque puedan producirse varias excepciones de tipos diferentes y no corresponder el mensaje (o tratamiento dado a la excepción) con el tipo exacto que se ha producido.

Especificando excepciones.

En el siguiente ejemplo mejoraremos el código anterior capturando distintos tipos de excepciones:

- ✓ TypeError: por si los operandos no permiten la división.
- ✓ ZeroDivisionError: por si el denominador es cero.
- ✓ Exception: para cualquier otro error que se pueda producir.

Veamos su implementación:

```
>>> def intdiv(a, b):
...     try:
...         result = a // b
...     except TypeError:
...         print('Check operands. Some of them seems strange...')
...     except ZeroDivisionError:
...         print('Please do not divide by zero...')
...     except Exception:
...         print('Ups. Something went wrong...')
...
>>> intdiv(3, 0)
Please do not divide by zero...
>>> intdiv(3, '0')
Check operands. Some of them seems strange...
```

Importante: las excepciones predefinidas en Python no hace falta importarlas previamente. Se pueden usar directamente.

Cubriendo más casos.

Python proporciona la cláusula “else” para saber que todo ha ido bien y que no se ha lanzado ninguna excepción. Esto es relevante a la hora de manejar los errores.

De igual modo, tenemos a nuestra disposición la cláusula “finally” que se ejecuta siempre, independientemente de si ha habido o no ha habido error.

Veamos un ejemplo de ambos:

```
>>> values = [4, 2, 7]
>>> user_index = 3
>>> try:
...     r = values[user_index]
... except IndexError:
...     print('Error: Index not in list')
... else:
...     print(f'Your wishes are my command: {r}')
... finally:
...     print('Have a good day!')
...
Error: Index not in list
Have a good day!
>>> user_index = 2
>>> try:
...     r = values[user_index]
... except IndexError:
...     print('Error: Index not in list')
... else:
...     print(f'Your wishes are my command: {r}')
... finally:
...     print('Have a good day!')
...
Your wishes are my command: 7
Have a good day!
```

7.2 Excepciones propias.

Python ofrece una gran cantidad de excepciones predefinidas. Hasta ahora hemos visto cómo gestionar y manejar este tipo de excepciones. Pero hay ocasiones en las que nos puede interesar crear nuestras propias excepciones. Para ello tendremos que crear una clase heredando de "Exception", la clase base para todas las excepciones.

Veamos un ejemplo en el que creamos una excepción propia controlando que el valor sea un número entero:

```
>>> class NotIntError(Exception):
...     pass
...
>>> values = (4, 7, 2.11, 9)
>>> for value in values:
...     if not isinstance(value, int):
...         raise NotIntError(value)
...
Traceback (most recent call last):
File "<stdin>", line 3, in <module>
__main__.NotIntError: 2.11
```

Hemos usado la sentencia "raise" para «elevar» esta excepción, que podría ser controlada en un nivel superior mediante un bloque try - except.

Nota: para crear una excepción propia basta con crear una clase vacía. No es imprescindible incluir código más allá de un pass.

Mensaje personalizado.

Podemos personalizar la excepción añadiendo un mensaje más informativo. Siguiendo el ejemplo anterior, veamos cómo introducimos esta información:

```
>>> class NotIntError(Exception):
...     def __init__(self, message='This module only works with integers. Sorry!'):
...         super().__init__(message)
...
>>> raise NotIntError()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.NotIntError: This module only works with integers. Sorry!
```

Podemos ir un paso más allá e incorporar en el mensaje el propio valor que está generando el error:

```
>>> class NotIntError(Exception):
...     def __init__(self, value, message='This module only works with integers. Sorry!'):
...         self.value = value
...         self.message = message
...         super().__init__(self.message)
...
...     def __str__(self):
...         return f'{self.value} -> {self.message}'
...
>>> raise NotIntError(2.11)
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
__main__.NotIntError: 2.11 -> This module only works with integers. Sorry!
```

7.3 Aserciones.

Si hablamos de control de errores hay que citar una sentencia en Python denominada "assert". Esta sentencia nos permite comprobar si se están cumpliendo las «expectativas» de nuestro programa, y en caso contrario, lanza una excepción informativa.

Su sintaxis es muy simple. Únicamente tendremos que indicar una expresión de comparación después de la sentencia:

```
>>> result = 10
>>> assert result > 0
>>> print(result)
10
```

En el caso de que la condición se cumpla, no sucede nada, y el programa continúa con su flujo normal. Esto es indicativo de que las expectativas que teníamos se han satisfecho. Sin embargo, si la condición que fijamos no se cumple, la aserción devuelve un error "AssertionError" y el programa interrumpe su ejecución:

```
>>> result = -1
>>> assert result > 0
-----
AssertionError Traceback (most recent call last)
<ipython-input-29-e2efe60b0c46> in <module>
----> 1 assert result > 0
AssertionError:
```

Podemos observar que la excepción que se lanza no contiene ningún mensaje informativo. Es posible personalizar este mensaje añadiendo un segundo elemento en la tupla de la aserción:

```
>>> assert result > 0, 'El resultado debe ser positivo'
-----
AssertionError Traceback (most recent call last)
<ipython-input-31-f58052ce672b> in <module>
----> 1 assert result > 0, 'El resultado debe ser positivo'
AssertionError: El resultado debe ser positivo
```