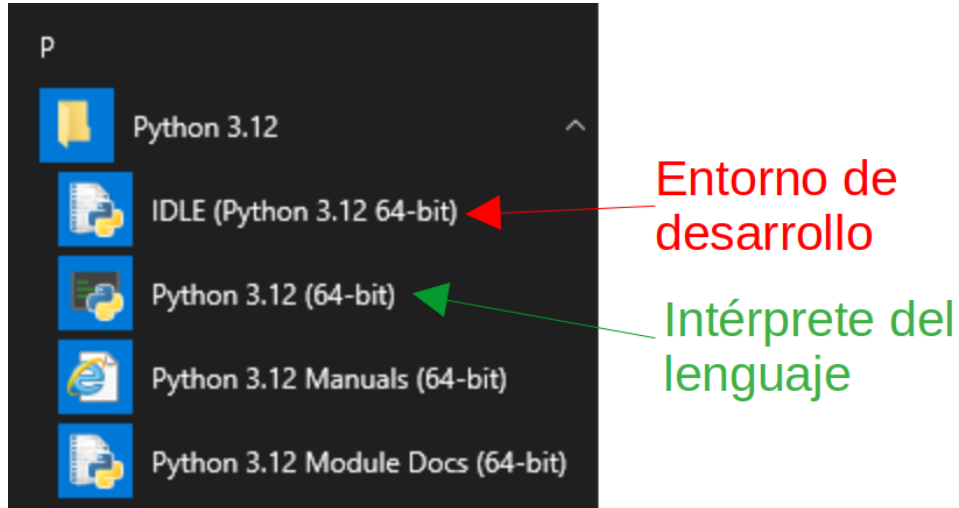


APUNTES DE PYTHON

0. INSTALACIÓN

Una vez descargado Python desde esta URL: <https://www.python.org/downloads>
Veremos esto en el menú de Inicio de Windows:



El primer icono nos permite abrir el entorno de desarrollo de Python (para crear programas y guardarlos como archivos con la extensión .py)

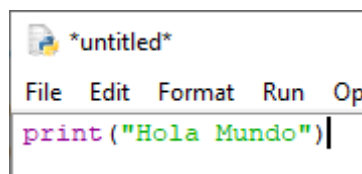
El segundo icono abre el intérprete, que permite ejecutar instrucciones de forma inmediata.

I.INTRODUCCIÓN

Primer programa

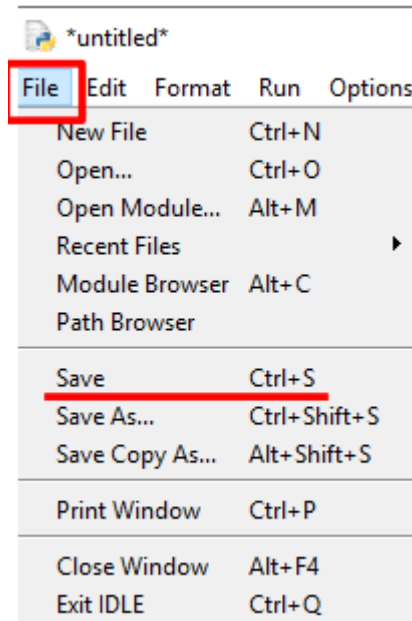
El primer programa que suele hacerse en cualquier lenguaje de programación es el “Hola Mundo”. En nuestro caso suponemos que tenemos instalado Python (el intérprete del lenguaje y el entorno de desarrollo o Idle).

Una vez descargado e instalado tenemos que ejecutar el Idle (es el entorno de desarrollo por defecto que incorpora Python) y escribimos lo siguiente:

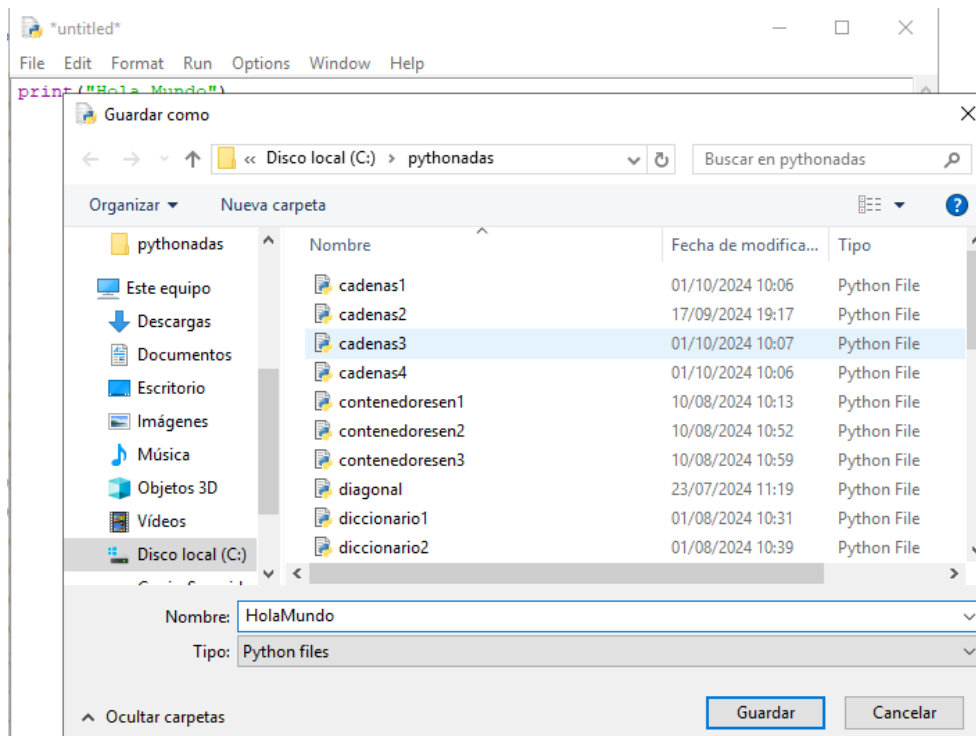


En la imagen superior vemos nuestro primer programa, aún sin guardar (por eso el archivo se llama untitled y aparece entre asteriscos).

A continuación guardamos el archivo (NO en la ubicación que ofrece por defecto) y lo ejecutamos con la opción del menú: *Run* (o *F5*). No puede ejecutarse un archivo que previamente no se haya guardado.

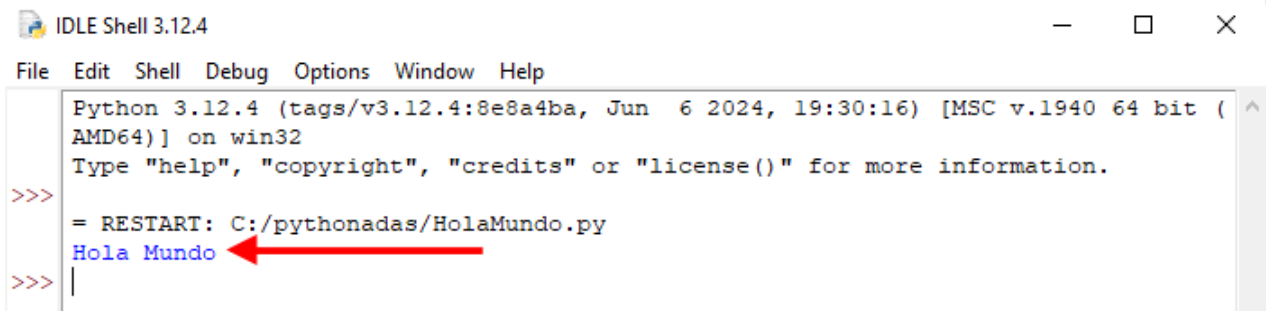


En la imagen superior vemos el archivo aún sin guardar (“*untitled”). Para guardarlo: *File* → *Save*.



En la imagen superior se ve cómo guardar el programa con el nombre *HolaMundo*. La extensión *.py* la añade, por defecto, *Idle*. Una vez que hayamos guardado el programa una vez podremos volver a guardar los cambios con *Ctrl* + *S*.

Cuando hayamos guardado el programa podemos ejecutarlo, bien mediante el menú:
Run → *Run Module* o con *F5*



A continuación se muestra el resultado de ejecutar el programa:

Comentarios

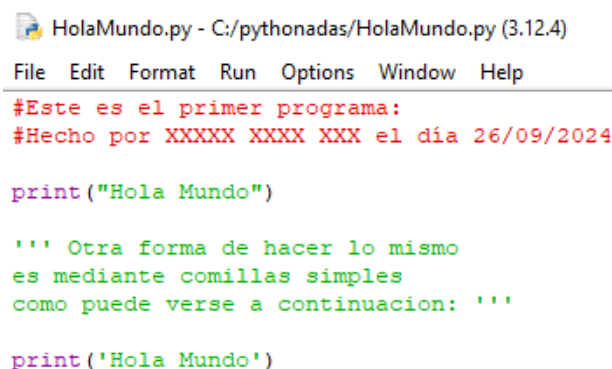
Lo normal es incluir, dentro del código de nuestros programas, comentarios (para nosotros o para otras personas que puedan ver ese programa).

Los comentarios pueden servir para:

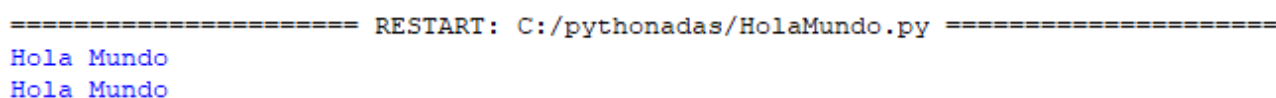
- Explicar lo que hace nuestro programa o partes de él.
- Identificar al creador del programa, la fecha de última modificación y la versión.
- Evitar que ciertas líneas de código se ejecuten: en ocasiones no queremos que se ejecuten ciertas líneas de código; para evitarlo podemos comentarlas, cuando queramos que se ejecuten tendremos que *descomentarlas*.

Una línea se comenta poniendo al comienzo un carácter especial, en el caso de Python es este: #
Si queremos comentar un conjunto de líneas podemos poner el símbolo # al comienzo de cada línea o bien poner tres comillas simples al comienzo y otras tres al final del bloque que queremos comentar:

Ejemplo de programa con los dos tipos de comentarios:



Y la ejecución de este programa:



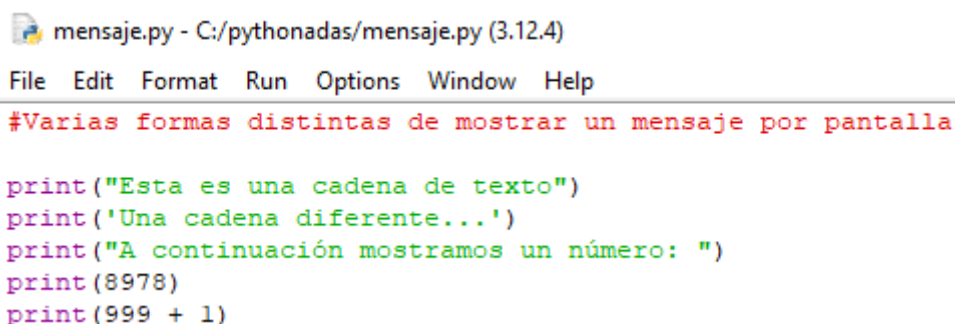
Interacción con el usuario

Las dos formas más básicas de interactuar con el usuario que utiliza el programa son:

- Mostrar mensajes por pantalla.
- Recibir datos del usuario.

La forma de mostrar mensajes por pantalla es usando `print ()` de alguna de estas formas:

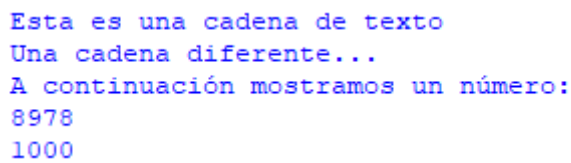
Donde vemos que a continuación de la palabra reservada `print` se abren unos paréntesis, dentro de los cuales se incluye el mensaje con comillas dobles o comillas simples, si queremos que el mensaje aparezca como una cadena de texto (ver el ejemplo anterior). Si queremos sacar un número no es necesario incluir comillas dentro de los paréntesis; si incluimos una operación con números, se resolverá antes y el resultado se muestra por pantalla:



```
mensaje.py - C:/pythonadas/mensaje.py (3.12.4)
File Edit Format Run Options Window Help
#Varias formas distintas de mostrar un mensaje por pantalla

print("Esta es una cadena de texto")
print('Una cadena diferente...')
print("A continuación mostramos un número: ")
print(8978)
print(999 + 1)
```

Al ejecutarlo veremos esto:



```
--
Esta es una cadena de texto
Una cadena diferente...
A continuación mostramos un número:
8978
1000
```

Cuando mezclamos en un `print` cadenas de texto y números tenemos que “preparar” la cadena que vamos a mostrar:

Si intentamos imprimir:

```
print("Este es el número: " + numero)
```

(suponiendo que “numero” es una variable que contiene un valor numérico) entonces obtendremos un error porque estamos intentado concatenar una cadena con un número (solo se pueden concatenar cadenas con otras cadenas), por lo tanto, tenemos que convertir el contenido de la variable numero en una cadena de esta manera: `str(numero)` donde `str` significa string:

```
print("Este es el número: " + str(numero))
```

Una forma alternativa de hacer lo anterior es la siguiente:

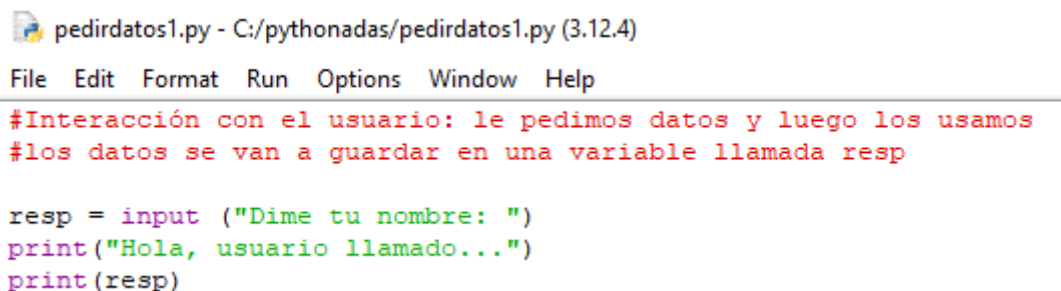
```
print(f"Este es el número: {numero}")
```

Donde la letra `f` (antes de las comillas y pegadas a ellas) hace que se interprete correctamente las variables que pongamos entre llaves: `{}` (ojo: NO comillas).

Por lo tanto las dos sentencias que se muestran a continuación son equivalentes:

```
print(f"Este es el número: {numero}")
print("Este es el número: " + str(numero))
```

Otra forma sencilla de interactuar con el usuario es solicitando al usuario datos que luego emplearemos en nuestro programa; los datos que el usuario nos proporciona los guardamos en una variable (una variable es un contenedor de datos):



```
pedirdatos1.py - C:/pythonadas/pedirdatos1.py (3.12.4)
File Edit Format Run Options Window Help
#Interacción con el usuario: le pedimos datos y luego los usamos
#los datos se van a guardar en una variable llamada resp

resp = input ("Dime tu nombre: ")
print("Hola, usuario llamado...")
print(resp)
```

Resultado de la ejecución:

```
= RESTART: C:/pythonadas/pedirdatos1.py
Dime tu nombre: Larry Bird
Hola, usuario llamado...
Larry Bird
```

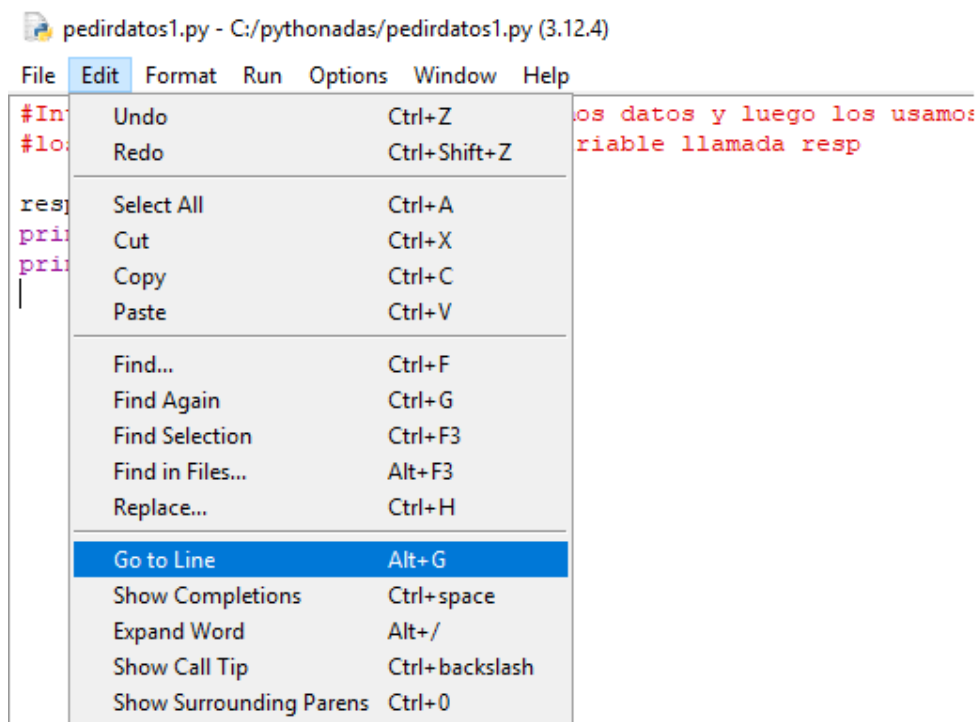
Las líneas de código en Python

Ir a un número de línea:

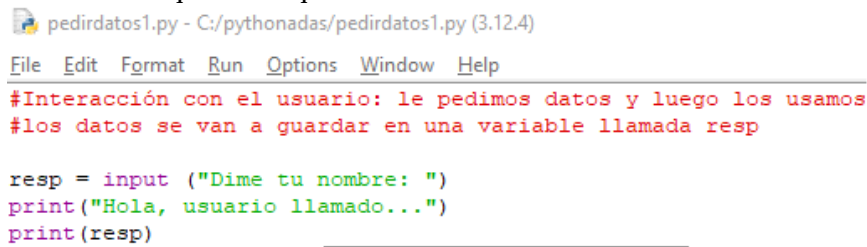
Es bastante útil poder ver las líneas del archivo en el que estamos escribiendo nuestro programa y poder desplazarnos a una línea concreta.

Mostrar o no las líneas no afecta en nada a la ejecución del programa.

Podemos desplazar a una determinada línea con la opción del menú: *Edit* → *Go to line* a continuación nos pregunta en qué línea del fichero queremos situarnos:

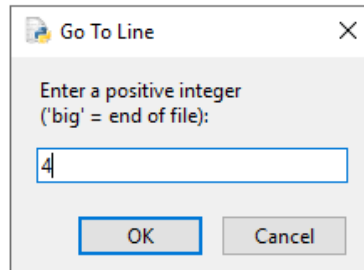


Y ahora indicamos en qué línea queremos situarnos:

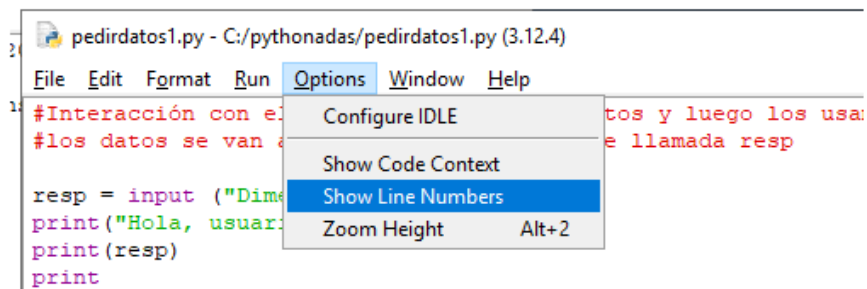


```
pedirdatos1.py - C:/pythonadas/pedirdatos1.py (3.12.4)
File Edit Format Run Options Window Help
#Interacción con el usuario: le pedimos datos y luego los usamos
#los datos se van a guardar en una variable llamada resp

resp = input("Dime tu nombre: ")
print("Hola, usuario llamado...")
print(resp)
```



Otra posibilidad es que, mediante la configuración del Idle, mostremos los números de línea:
Options → Show Line Numbers



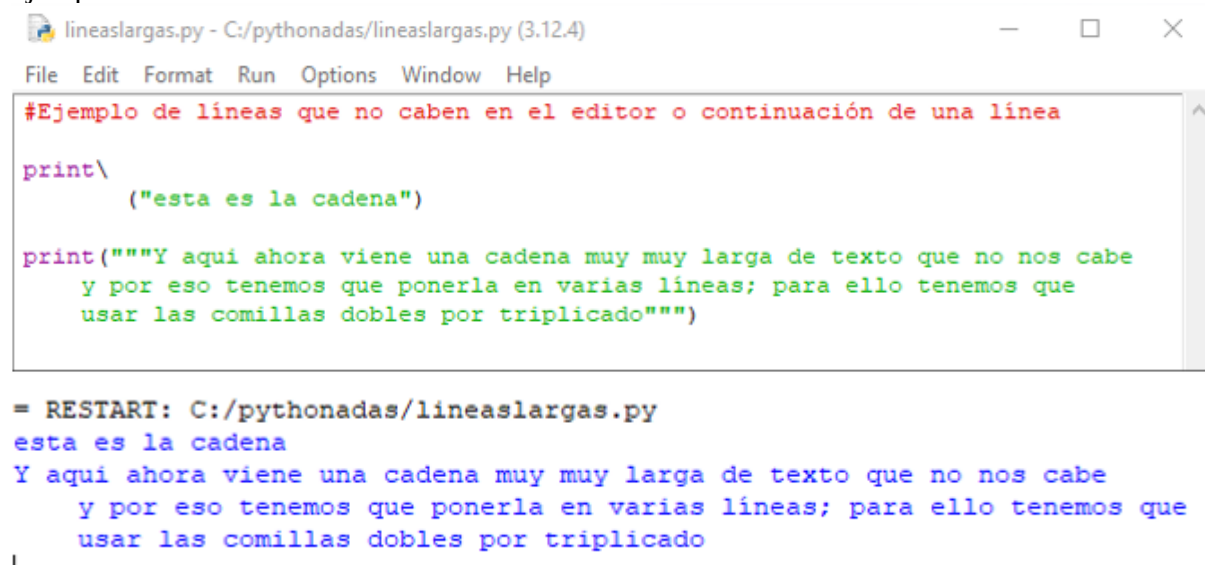
```
pedirdatos1.py - C:/pythonadas/pedirdatos1.py (3.12.4)
File Edit Format Run Options Window Help
#Interacción con el usuario: le pedimos datos y luego los usamos
#los datos se van a guardar en una variable llamada resp

resp = input("Dime tu nombre: ")
print("Hola, usuario llamado...")
print(resp)
```

En ocasiones puede haber líneas de código muy largas que no quepan en el editor; podemos escribir una línea debajo de otra usando el carácter *backslash* (o contrabarra): \

En el caso de una cadena de texto podemos usar tres comillas dobles al comienzo y al final de la cadena.

Ejemplo:



```
lineaslargas.py - C:/pythonadas/lineaslargas.py (3.12.4)
File Edit Format Run Options Window Help
#Ejemplo de líneas que no caben en el editor o continuación de una línea

print\
    ("esta es la cadena")

print("""Y aquí ahora viene una cadena muy muy larga de texto que no nos cabe
y por eso tenemos que ponerla en varias líneas; para ello tenemos que
usar las comillas dobles por triplicado""")

= RESTART: C:/pythonadas/lineaslargas.py
esta es la cadena
Y aquí ahora viene una cadena muy muy larga de texto que no nos cabe
    y por eso tenemos que ponerla en varias líneas; para ello tenemos que
    usar las comillas dobles por triplicado
```

Palabras reservadas

Hay ciertas palabras que no pueden usarse en Python (por ejemplo, para nombrar una variable), por estar reservadas para el lenguaje.

Podemos hacer que el intérprete de Python nos muestre el listado de palabras reservadas:

```
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
```

Estas son las palabras que no podemos asignar a ningún objeto definido por nosotros.

Tipos de datos

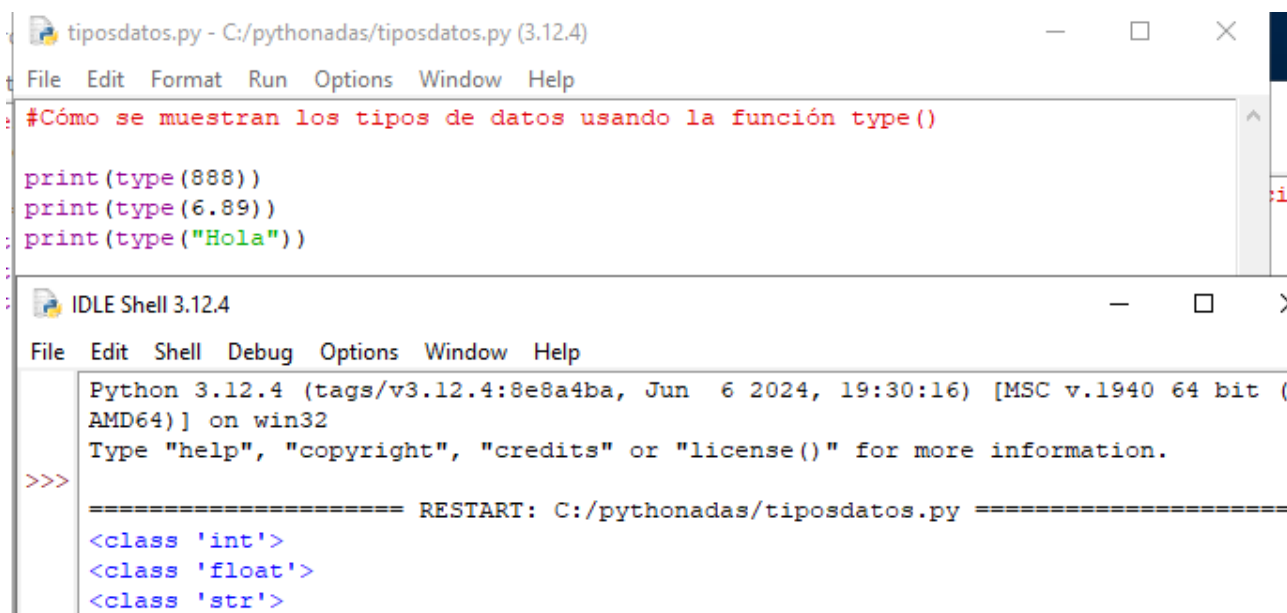
Las variables que almacenan datos tienen que adaptarse al tipo de dato que reciben (lo hacen de forma automática), pero no es lo mismo almacenar un dato numérico que un cadena de caracteres (la cantidad de memoria que se usa es distinta en cada caso). Incluso para datos numéricos tampoco se reserva la misma memoria para un entero que para un número real.

Según https://www.w3schools.com/python/python_datatypes.asp estos son los tipos de datos en Python:

Tipo	Abreviatura	Uso
Texto (string)	str	Cadenas de texto
Numéricos	int	Números enteros
	float	Números con decimales
	complex	Números complejos
Colecciones (secuencias)	list	Listados
	tuple	Tuplas
	range	Rangos
Mapeados	dict	Diccionarios
Conjuntos	set	Conjunto de datos
	frozenset	Conjunto de datos no modificable
Booleanos	bool	Valores lógicos: 0/1; True/False
Binarios	bytes	
	bytearray	
	memoryview	
Ningún tipo	NoneType	None. Ausencia de valor.

Pregunta: ¿por qué es importante indicar la ausencia de un valor?

Puede mostrarse el tipo de dato de una variable usando *type()*



The screenshot shows two windows from the Python IDLE 3.12.4 application. The top window, titled 'tiposdatos.py - C:/pythonadas/tiposdatos.py (3.12.4)', contains the following Python code:

```
#Cómo se muestran los tipos de datos usando la función type()

print(type(888))
print(type(6.89))
print(type("Hola"))
```

The bottom window, titled 'IDLE Shell 3.12.4', shows the output of running the script. It displays the Python version and architecture, followed by the output of the `print` statements:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun 6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/pythonadas/tiposdatos.py =====
<class 'int'>
<class 'float'>
<class 'str'>
```

Hay que tener en cuenta que en Python cada valor es un objeto. A cada objeto le corresponde un tipo de dato: categoría de datos a la que pertenece el objeto, esto determina las propiedades del objeto. El valor del objeto es el dato que representa. “Hola Mundo” es un objeto que pertenece al tipo de datos string (o str).

Constantes y variables

En cualquier lenguaje de programación habrá valores que queremos modificar y otros que no cambiaremos a lo largo del programa.

Los valores se almacenan en unas posiciones de memoria a las que se da un nombre para poder usarla dentro del programa; es nombre es un etiqueta, no el valor de la variable.

Una variables es un contenedor de datos de un determinado tipo. Su contenido (o valor) puede cambiar durante la ejecución del programa.

Una constante es un valor que nunca cambia; ejemplo: cuando usamos un número directamente en el programa estamos usando una constante.

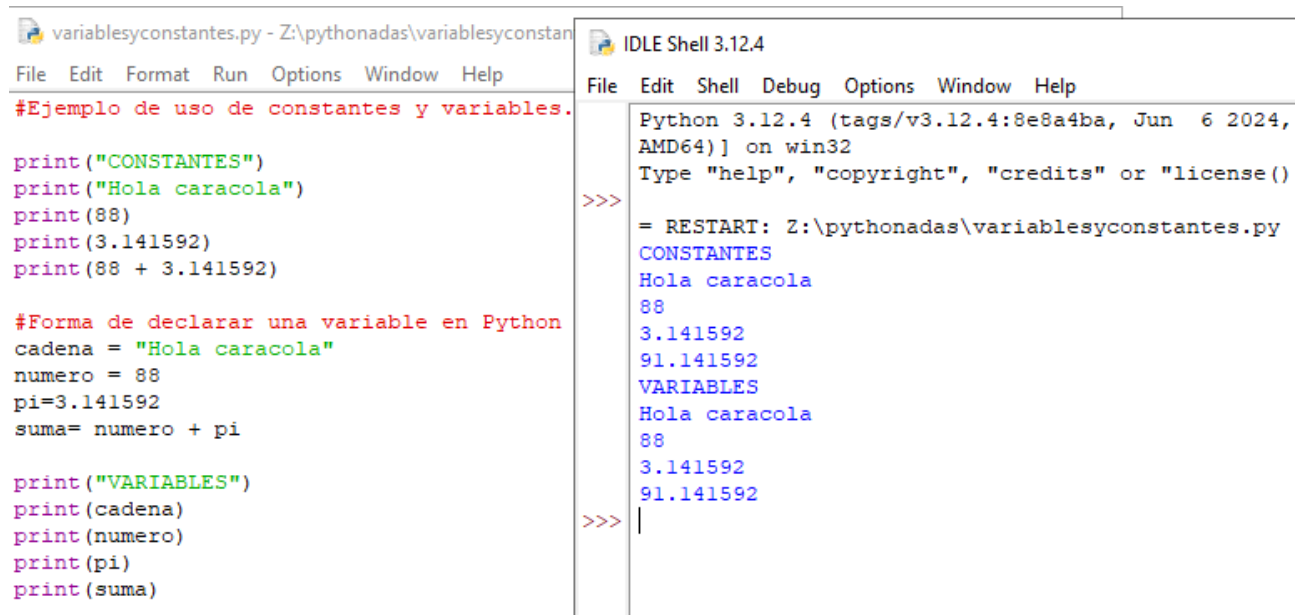
En general, se considera una mala práctica de programación el uso de constantes; casi siempre es preferible usar variables.

La forma de declarar una variable es poniendo su nombre y a continuación el valor que queremos asignarle, usando un signo de igualdad (“=”).

Hay pocas normas para declarar variables:

- 1) No espacios.
- 2) Solo se admiten letras, números y el guion bajo “_”.
- 3) No se puede comenzar el nombre de la variable con un número, sí con un guion bajo pero tiene un significado especial.
- 4) No usar palabras reservadas por el lenguaje de programación.

Ejemplo de uso de variables y constantes:



The screenshot shows a Python IDE with two windows. The left window, titled 'variablesyconstantes.py', contains the following code:

```
#Ejemplo de uso de constantes y variables.

print("CONSTANTES")
print("Hola caracola")
print(88)
print(3.141592)
print(88 + 3.141592)

#Forma de declarar una variable en Python
cadena = "Hola caracola"
numero = 88
pi=3.141592
suma= numero + pi

print("VARIABLES")
print(cadena)
print(numero)
print(pi)
print(suma)
```

The right window, titled 'IDLE Shell 3.12.4', shows the output of the script:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun 6 2024,
AMD64) on win32
Type "help", "copyright", "credits" or "license()"
>>>
= RESTART: Z:\pythonadas\variablesyconstantes.py
CONSTANTES
Hola caracola
88
3.141592
91.141592
VARIABLES
Hola caracola
88
3.141592
91.141592
>>> |
```

Pregunta: ¿Pueden mezclarse en una expresión aritmética variables y constantes?

Operadores aritméticos

Para hacer operaciones de tipo aritmético (sumas, restas, divisiones,...) se necesitan los operandos y el símbolo que indica la operación que haremos; estos símbolos son los operadores aritméticos y en Python son los siguientes:

Operador	Significado	Ejemplo	Resultado
+	Suma	2 + 2	4
-	Resta	4 - 1	3
*	Multiplicación	3 * 5	15
/	División	13 / 8	1.625
//	División entera	13 // 8	1
%	Módulo o resto	13 % 8	5
**	Exponente	2 ** 4	16

Ejercicios: Hacer un programa que muestre un ejemplo de cada operador. Usar variables y constantes.

Existen funciones matemáticas más complejas (logaritmos, raíz cuadrada,...) pero requieren importar el módulo apropiado.

Existe una forma especial de usar el operador incremento cuando trabajamos con variables:

`x += 1` que es equivalente a `x = x + 1`

De igual modo puede usarse con el operador de resta.

Pregunta: ¿puede usarse algún otro operador de esta forma?

Operadores de comparación

Otra categoría de operadores que se usan muy a menudo son los operadores de comparación; en general sirven para comparar dos operandos (uno a cada lado del operador) y el resultado es siempre True o False (Verdadero o Falso).

Operador	Significado	Ejemplo	Resultado
>	Mayor que	100 > 10	True
<	Menor que	100 < 10	False
>=	Mayor o igual que	2 <= 2	True
<=	Menor o igual que	1 <= 4	True
==	Equal	6 == 9	False
!=	Distinto	3 != 2	True

Es muy habitual usar estos operadores en estructuras de programación de tipo bifurcación (if-then-else) o repetitivas (bucles: for, while).

Pueden combinarse estos operadores con los aritméticos vistos en el punto anterior.

Operadores lógicos

Otra categoría de operadores que usan como operandos expresiones, variables o constantes que evalúan a True o False. El resultado será siempre True o False.

Operador	Significado	Ejemplo	Resultado
and	Y	True and True	True
or	O	True or False	True
not	no	Not True	False

Es muy habitual usar estos operadores junto con los de comparación.

Ejs.:

(5 < 7) and (8 <= 1) → True and False = False

(5 < 7) or (8 <= 1) → True or False = True

not (5 < 7) or (8 <= 1) → False or False = False

2. ESTRUCTURAS DE CONTROL

2.1 Sentencias/Instrucciones Condicionales (Conditional statements. Statement = sentencia, instrucción)

En ocasiones interesa interrumpir el flujo normal de ejecución del programa o, dicho de otro modo: queremos ejecutar algunas instrucciones solo si se produce un hecho, o se dé una condición.

If-Else:

La instrucción más habitual para hacer esto es If-Then-Else que significa Si-Entonces-Si no. Es decir: comprobamos si se da una condición, si se cumple la condición que estamos comprobando, se ejecutan unas líneas del programa y, en caso contrario, se ejecutan otras alternativas (o quizá ninguna) pero no se van a ejecutar nunca las líneas que hay dentro del Si y las que hay dentro del Else.

Atención a los dos puntos después de la condición del if y después del else.

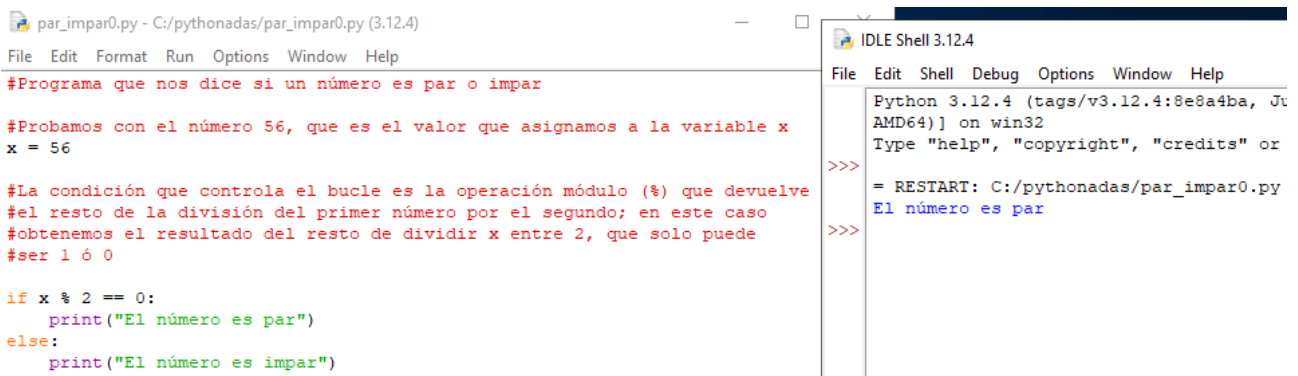
Ej.:

Pseudocódigo

Si (X es divisible por 2) entonces
 mostrar (X es un número par)
Si no
 mostrar (X es un número impar)
Fin Si

Código

```
if x % 2 == 0:  
    print("El número es par")  
else:  
    print("El número es impar")
```



Un esquema (diagrama de flujo) para ver cómo funciona esta instrucción:

Pseudocódigo y diagramas de flujo

Estructuras de Selección

if (condición)

Tarea a

.....

Tarea x

Else

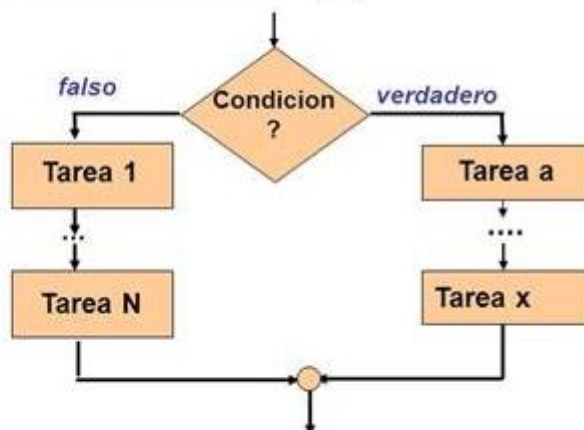
Tarea 1

.....

Tarea N

end if

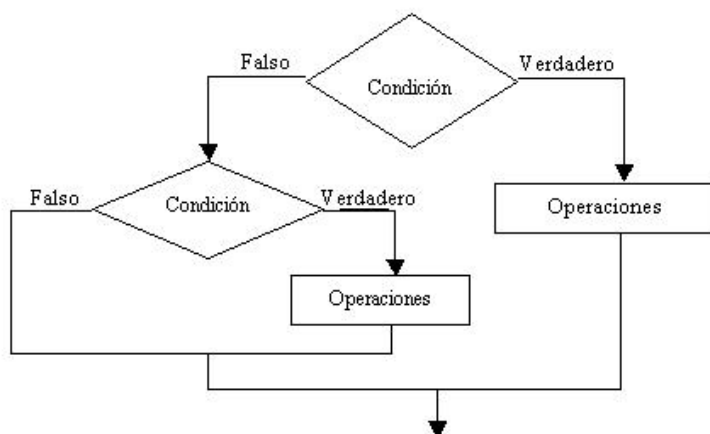
*Representación
en pseudocódigo*



*Representación en
Diagrama de Flujo*

Las sentencias condicionales if-else pueden anidarse; es decir: se puede poner otro if-else dentro del if, dentro del else o dentro de ambos.

A continuación, se muestra un diagrama que tiene un if-else con otro if-else anidado en el caso de la que condición sea falsa. A la derecha se muestra el código que correspondería con el diagrama de flujo de la izquierda.

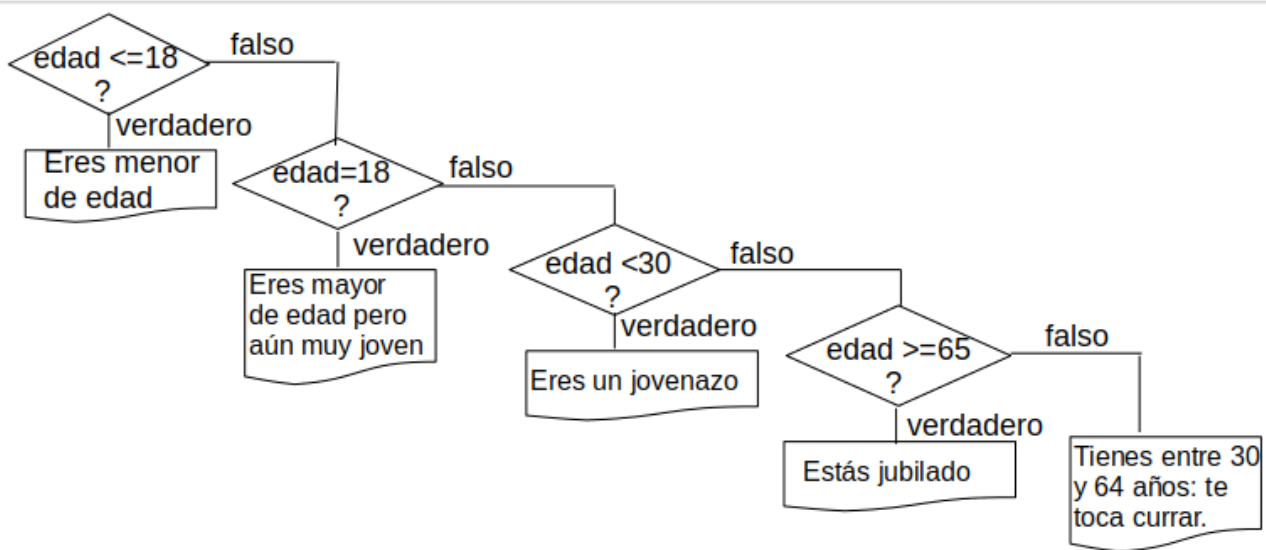


if condición:
Operaciones
else:
if condición:
Operaciones

Ejemplo de sentencias if-else anidadas:

```
if edad <= 18:
    print("Eres menor de edad")
else:
    if edad == 18:
        print("Ya eres mayor de edad, aunque aún muy joven")
    else:
        if edad < 30:
            print("Eres un jovenazo")
        else:
            if edad >= 65:
                print("Estás jubilado")
            else:
                print("Tienes entre 30 y 64 años: te toca currar")
```

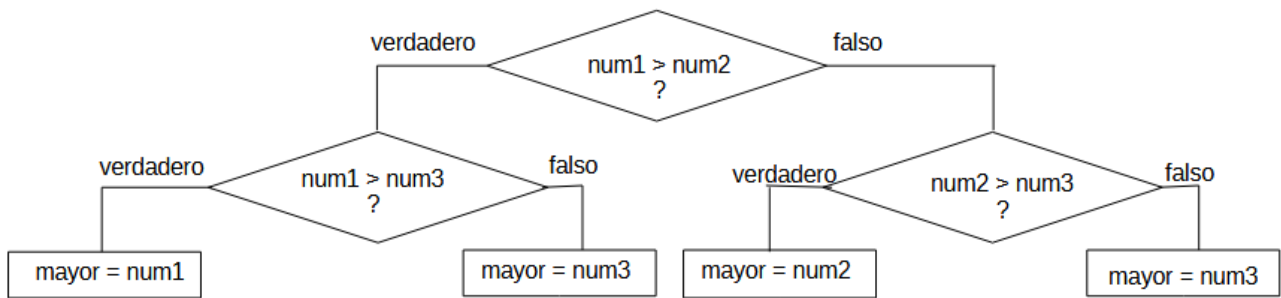
El diagrama de flujo que corresponde al código superior sería este:



Otra forma de hacer lo anterior es mediante sentencias **elif** que son la suma de else con un if:

```
if edad < 18:
    print("Eres menor de edad")
elif edad == 18:
    print("Ya eres mayor de edad, aunque aún muy joven")
elif edad < 30:
    print("Eres un jovenazo")
elif edad >= 65:
    print("Estás jubilado")
else:
    print("Tienes entre 30 y 64 años: te toca currar")
```

Un posible algoritmo para, dados tres números distintos, decir cuál es el mayor sería el siguiente:



```
SI num1 > num2 ENTONCES
    SI num1 > num3 ENTONCES
        mayor = num1
    ELSE
        mayor = num3
    FIN SI
ELSE
    SI num2 > num3 ENTONCES
        mayor = num2
    ELSE:
        mayor = num3
    FIN SI
FIN SI
```

Duda: ¿cuántas veces tendríamos que ejecutar el programa que implemente este algoritmo para probar todas las posibles combinaciones?

¿Cómo podría hacerse lo anterior, pero usando condiciones compuestas en el if-else?
Una forma de hacerlo sería lo siguiente:

```
if num1 > num2 and num1 > num3:
    mayor = num1
if num2 > num1 and num2 > num3:
    mayor = num2
if num3 > num1 and num3 > num2:
    mayor = num3
```

En ocasiones tenemos que elegir una sola opción de entre muchas posibilidades; en este caso se puede usar una serie de if-else, como en el ejemplo: pedimos un número entre [1-7] al usuario e imprimimos el día de la semana correspondiente a ese número. El pseudocódigo sería:

Versión 1

```
Pedir número
si número == 1 entonces
    imprimir "lunes"
si número == 2 entonces
    imprimir "martes"
si número == 3 entonces
    imprimir "miércoles"
si número == 4 entonces
    imprimir "jueves"
si número == 5 entonces
    imprimir "viernes"
si número == 6 entonces
    imprimir "sábado"
si número == 7 entonces
    imprimir "domingo"
si día < 1 OR día > 7
    imprimir "Error"
```

Versión 2

```
Pedir número
si número == 1 entonces
    imprimir "lunes"
si no
    si número == 2 entonces
        imprimir "martes"
    si no
        si número == 3 entonces
            imprimir "miércoles"
        si no
            si número == 4 entonces
                imprimir "jueves"
            si no
                si número == 5 entonces
                    imprimir "viernes"
                si no
                    si número == 6 entonces
                        imprimir "sábado"
                    si no
                        si número == 7 entonces
                            imprimir "domingo"
                        si no
                            imprimir "Error"
```

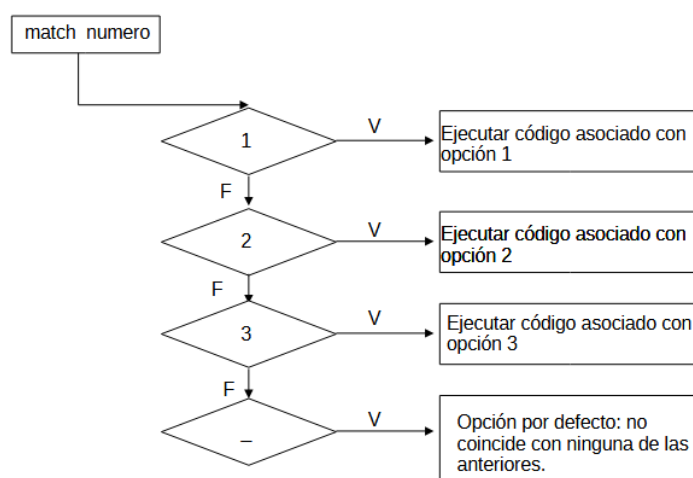
Duda: ¿Cómo se podría hacer esto mismo pero con un único "imprimir"?

Duda: ¿Qué dos posibilidades habría de programar la versión 2?

match-case:

Una forma más habitual de elegir una opción de entre muchas es usar una estructura llamada match-case que se basa en la coincidencia de un valor dado con el contenido de una variable (la variable que controla el match-case). Ojo: esta estructura programación existe desde la versión 3.10 del lenguaje de programación.

El diagrama de flujo es:



Se compara el contenido de la variable con varios valores (o patrones) y se ejecutarán las líneas de código que coinciden con el valor dado.

Se puede incluir una opción por defecto: cuando el valor de la variable no coincide con ninguno de los anteriores. En caso de añadir una opción por defecto tiene que ir siempre en último lugar.

El caso anterior quedaría así:

```
2 numero = 17
3 match numero:
4     case 1:
5         print("Número uno")
6     case 2:
7         print("Número dos")
8     case 3:
9         print("Número tres")
10    case _:
11        print("Es una opción diferente a uno, dos o tres")
```

```
| Es una opción diferente a uno, dos o tres
|
```

2.2 Sentencias/Instrucciones Repetitivas o bucles (Loop statements).

Un bucle es una estructura de programación que sirve para repetir un conjunto de líneas (o sentencias) varias veces, mientras se cumpla una determinada condición.

A cada repetición del bucle se le llama “iteración”. El número de iteraciones o repeticiones de un bucle depende del valor de una variable llamada variable de control, por lo tanto, un bucle tendrá siempre una variable de control que determinará cuántas veces se ejecutan las sentencias del interior del bucle. El número de repeticiones (o iteraciones) del bucle puede conocerse antes de que se ejecute el bucle, puede no conocerse (y dependerá de algún evento: las respuestas que da el usuario, por ejemplo); también puede ocurrir que el bucle no llegue a ejecutarse ninguna vez o que se ejecute un número (en principio) infinito de veces. La variable de control del bucle tiene que cambiar en el interior del bucle para que no se siga ejecutando indefinidamente.

En Python hay dos tipos de bucles: bucles for y bucles while.

Bucles for:

En los bucles for sabemos desde el principio cuántas veces se va a repetir (sabemos cuántas iteraciones hará).

Hay dos formas de hacer bucles for:

1) Bucles for que recorren un conjunto de valores que se indica expresamente. Tienen la forma siguiente:

```
for variable in conjunto_de_valores:
    líneas del interior del bucle
```

Ejemplo:

```
#Bucle for que recorre los días de la semana.
#Los días de la semana están definidos como un conjunto.

for dia in ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"]:
    print(dia)
```


Al ejecutarlo obtendríamos:

```
===== RESTART: G:/Curso24-25/INS/Python/pythonadas/for_dias.py ==
Lunes
Martes
Miércoles
Jueves
Viernes
Sábado
Domingo
```

Donde vemos que: la variable “dia” va almacenando, a cada iteración de bucle un valor del conjunto, desde el primero hasta el final: en la primera iteración del bucle la variable “dia” tiene el valor “Lunes”; en la segunda iteración dia = “Martes”, en la tercera: dia=“Miércoles” y así hasta llegar al valor final del conjunto de valores que queremos recorrer. Podemos definir el conjunto de valores creando, previamente, lo que se conoce como una lista; un par de ejemplos:

Definimos los siguientes conjuntos de valores:

```
Star_wars = ["Han Solo", "Luke Skywalker", "Chewbacca", "R2D2", "C3PO", "Darth Vader"]

Celtics = ["Larry Bird", "Kevin McHale", "Robert Parish", "Dennis Johnson", "Danny Ainge", "Cedric Maxwell", "Scott Wedman", "Ray Williams", "Greg Kite", "M.L. Carr", "Rick Carlisle", "Quinn Buckner"]

#Otro ejemplo de bucle for donde se definen previamente los valores mediante listas

Star_wars = ["Han Solo", "Luke Skywalker", "Chewbacca", "R2D2", "C3PO", "Darth Vader", "Maestro Yoda"]
Celtics = ["Larry Bird", "Kevin McHale", "Robert Parish", "Dennis Johnson", "Danny Ainge", "Cedric Maxwell"]

print("~~~ Personajes de Star Wars: ~~~")
for personaje in Star_wars:
    print(personaje)

print("$$$ Plantilla Boston Celtics 1984/85: $$$")
for personaje in Star_wars:
    print(personaje)
```

2) Bucles for que recorren un rango de valores, de los cuales se indican los valores inicial y final (y en ocasiones también el incremento).

Tienen la forma siguiente:

```
for variable in rango(valor_inicial,valor_final, incremento):
    líneas del interior del bucle
```

Donde pueden faltar algunos de elementos, lo que da lugar a tres posibilidades:

- a) for range(max): se recorren los valores desde 0 hasta *max*-1
- b) for range(min, max): se recorren los valores consecutivos desde *min* hasta *max*-1
- c) for range(min, max, step): se recorren los valores desde *min* hasta *max*-1 con incrementos de *step* en *step*

Hay que tener en cuenta la siguiente peculiaridad: la variable recorrerá los valores en el rango desde *valor_inicial* hasta *valor_final* – 1.

Ejemplo del tipo a:

```
#Ejemplo de bucle for que recorre un rango de valores: desde x=0 hasta x=5-1.  
#En los apuntes se ha llamado tipo a:
```

```
for x in range(5):  
    print(x)
```

Resultado de la ejecución:

```
===== RESTART: G:\Curso24-25\INS\Python\pythonadas\for_rango_a.py :  
0  
1  
2  
3  
4
```

Ejemplo del tipo b:

```
#Ejemplo de bucle for que recorre un rango de valores: desde x=1 hasta x=10-1.  
#En los apuntes se ha llamado tipo b:
```

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

Y lo que veríamos por pantalla:

```
===== RESTART: G:/Curso24-25/INS/Python/pythonadas/for_rango_b.py  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

Ejemplo del tipo c:

```
#Ejemplo de bucle for que recorre un rango de valores: desde x=0 hasta x=11-1 con incrementos de 2 en 2.  
#En los apuntes se ha llamado tipo c:
```

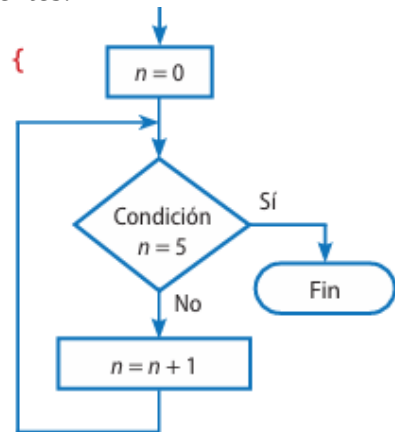
```
for x in range(0,11,2):  
    print(x)
```

Al ejecutarlo obtenemos esto:

```
===== RESTART: G:/Curso24-25/INS/Python/pythonadas/for_rango_c.py  
0  
2  
4  
6  
8  
10  
.
```

El pseudocódigo y el diagrama de flujo de un bucle for son los siguientes:

```
for(inicialización; condición; incremento o decremento ) {  
    sentencia 1;  
    ...  
    sentencia n;  
}
```



Es decir: el bucle for incluye dentro de su estructura la condición y el contador que interviene en la condición.

Existen tres secciones:

- Inicialización: para asignar un valor inicial a la variable que controla el bucle. Se ejecuta sólo una vez, cuando empieza el bucle, después el bucle se repite tantas veces como se cumpla la condición.
- Condición: es una comprobación que se hace sobre la variable que controla el bucle.
- Incremento o decremento: la variable que controla el bucle tiene que cambiar su valor, aumentando o disminuyendo, partiendo del valor inicial y hasta el valor final que marca el final del bucle.

Bucles for anidados:

Es posible que entre los comandos que queramos ejecutar dentro de un bucle esté otro bucle; en este caso hay que tener en cuenta que para cada iteración del bucle externo se harán todas las iteraciones del bucle interno.

Con los bucles anidados recorreremos estructuras en dos niveles; ejemplo: filas y columnas.

Ejemplo:

#Ejemplo de bucles for anidados: para cada elemento de los que recorre el bucle exterior se recorren todos los elementos del bucle interior.

```
for i in range(1, 4): # Bucle para 1, 2, 3  
    for j in range(1, 4): # Bucle para 1, 2, 3  
        print(i, j)
```

Y el resultado sería:

```
1 1  
1 2  
1 3  
2 1  
2 2  
2 3  
3 1  
3 2  
3 3
```

Otro ejemplo:

#Ejemplo de bucles for anidados: para cada elemento de los que recorre el bucle
#exterior se recorren todos los elementos del bucle interior.
#Suponemos un hotel con 5 plantas y cuatro habitaciones (A, B, C, D) en cada planta

```
habitaciones = ["A", "B", "C", "D"]
for i in range(1,6):
    print(f"Piso: {i}")
    for j in habitaciones:
        print(f"Habitación: {j}", end="\t") # Mostrar la suma de índices
    print() # Salto de línea después de cada fila
```

Al ejecutarlo:

```
Piso: 1
Habitación:  A  Habitación:  B  Habitación:  C  Habitación:  D
Piso: 2
Habitación:  A  Habitación:  B  Habitación:  C  Habitación:  D
Piso: 3
Habitación:  A  Habitación:  B  Habitación:  C  Habitación:  D
Piso: 4
Habitación:  A  Habitación:  B  Habitación:  C  Habitación:  D
Piso: 5
Habitación:  A  Habitación:  B  Habitación:  C  Habitación:  D
```

Bucles while:

En los bucles while no sabemos desde el principio cuántas veces se va a repetir: que se sigan ejecutando o no dependerá de una variable, que es la que controla el bucle, y que tiene que:

- 1) Inicializarse fuera del bucle.
- 2) Modificarse dentro del bucle.

Puede no llegar a ejecutarse ninguna vez o puede ejecutarse un número variable de veces (dependiendo del valor que la variable tome dentro del bucle) o puede ejecutarse indefinidamente (bucle infinito: se seguirá ejecutando hasta que, desde fuera del programa se cierre el proceso).

La forma genérica de este bucle es la siguiente:

while [expresión]: código a ejecutar.

El bucle se ejecutará mientras se cumpla la condición; si la condición no se cumple antes de ejecutar el bucle, no se llegará a entrar en éste; si la condición se cumple antes de ejecutar el bucle, se ejecutará hasta que la condición deje de cumplirse, para que eso ocurra tenemos que modificar la variable dentro del bucle.

Ejemplo:

```
#Bucle While que imprime los números del 1 a 10
x = 1
while x <= 10:
    print(f"{x}")
    x += 1

print("Hasta aquí la cuenta!")
```

Vemos que la condición de entrada al bucle se cumple, porque $1 \leq 10$; dentro del bucle vamos incrementando la variable x que es la que controla la ejecución del bucle que se repetirá hasta que deje de cumplirse que $x \leq 10$; cuando $x = 10$, aún se cumple, cuando $x = 11$, ya no se cumple y nos salimos del bucle.

El resultado de ejecutar el código anterior es el siguiente:

```
1
2
3
4
5
6
7
8
9
10
Hasta aquí la cuenta!
```

En un bucle Python puede haber una condición compuesta para controlar el bucle, en este caso el bucle seguirá ejecutándose mientras la condición compuesta sea verdadera. Ejemplos:

El bucle siguiente seguirá ejecutándose mientras la condición compuesta sea verdadera, como la condición está formada por dos partes unidas por un OR basta con que se cumpla una de las dos para que el bucle siga ejecutándose.

```
while x <= 3 or acierto = true:
```

A continuación se muestra un caso en que tienen que ciertas ambas condiciones, porque están unidas por un AND, si una de las dos partes de la condición compuesta deja de ser verdadera, el bucle dejará de ejecutarse:

```
while x <= true and acierto = false:
```

Otra forma de terminar un bucle de forma inmediata, sin esperar a que se evalúe la condición, es usando la palabra reservada **break** dentro del bucle; en cuanto se llegue a ella se dejará de ejecutar el bucle.

Ejemplo:

```
#Ejemplo de bucle que deja de ejecutar al encontrar un break
for i in range (1,100):
    print(i)
    break
```

Al ejecutar este bucle el resultado es el siguiente:

```
===== RESTART: C:/Python/Clase/ejercicioswhile/while_4break.py =====
1
```

Lo normal es que el break esté asociado a un if y que se ejecute solo en circunstancias especiales que tenemos que controlar nosotros.

Ejemplo:

El programa sigue preguntando hasta que respondamos “q”:

```
pregunta = "¿Y tú de quién eres?"

while True:
    resp = input(pregunta)
    if resp == "q":
        break
    print("Así que eres de " + resp + " mira tú qué bien")

print("Ea, se acabó de preguntar, señora!")
exit()
```

Al ejecutarlo:

```
¿Y tú de quién eres?Marujita
Así que eres de Marujita mira tú qué bien
¿Y tú de quién eres?Manolita
Así que eres de Manolita mira tú qué bien
¿Y tú de quién eres?Rebujita
Así que eres de Rebujita mira tú qué bien
¿Y tú de quién eres?q
Ea, se acabó de preguntar, señora!
```

Esta idea se usa bastante para los menús que controlan la ejecución de un programa: la ejecución del bucle se controla con un while true (en principio, bucle infinito), hasta que el usuario introduce una opción para salir del programa (normalmente se asocia a “q” por “quit”).

A veces podemos querer saltarnos una (pero sólo una) ejecución del bucle, para eso usamos **continue** cuando se encuentre esta palabra el bucle pasará a la siguiente ejecución. Ejemplo:

#Ejemplo de bucle while con un continue

```
i=1
while i <= 15:
    if i == 13:
        i += 1
        continue
    print(i)
    i += 1
```

Si lo ejecutamos:

```
1
2
3
4
5
6
7
8
9
10
11
12
14
15
|
```

Duda: ¿qué pasaría si quitáramos la línea `i+=1` que está dentro del `if`?

Realmente tiene más sentido usarlo en un bucle `for`, más que en un `while`.

```
equipos = ["Lakers", "Celtics", "Pistons", "Atlanta", "Sixers"]

for x in equipos:
    if x == "Pistons":
        continue
    print("Me gustan los " + x)
```

Al ejecutarlo:

```
Me gustan los Lakers
Me gustan los Celtics
Me gustan los Atlanta
Me gustan los Sixers
```

Como vemos, el `continue` también debe ir asociado a un `if`.

3. FUNCIONES

Un principio fundamental a la hora de hacer programas es intentar que éstos no sean enormes cadenas de instrucciones sin una estructura. Es preferible dividir y encapsular las líneas de código en unos subprogramas llamados funciones: esto es un principio fundamental de la programación estructurada.

Una función consiste en una serie de líneas de código con un nombre que se definen en un programa y que posteriormente puede llamarse desde cualquier parte del programa.

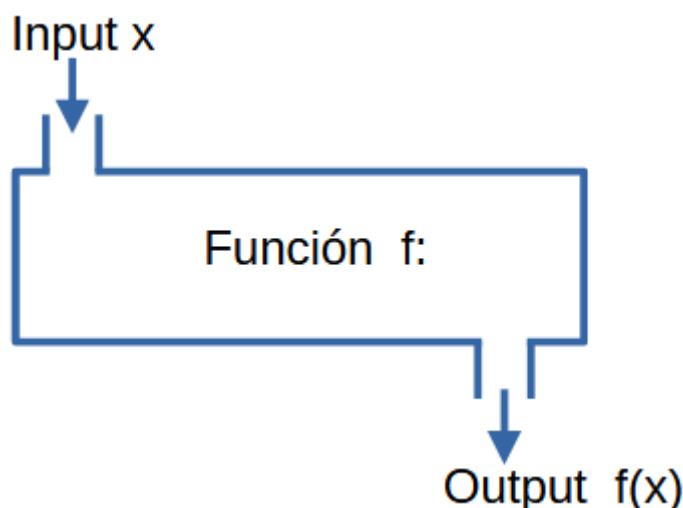
El principio fundamental a la hora de definir funciones es que éstas tienen que hacer una única cosa.

Hay funciones que necesitan que se les pasen unos datos para poder hacer su tarea; otras no necesitan que se les proporcione ningún dato.

Hay muchas ventajas relacionadas con el uso de funciones:

1. Modularidad: al dividir el código en módulos (o sea: funciones) los programas son más fáciles de entender, mantener y depurar, porque cada función hace una tarea concreta (y solo una!).
2. Reutilizar el código: el código de la función se escribe una vez, se prueba, y, cuando funciona correctamente, puede reutilizarse (llamarse) desde cualquier parte del programa.
3. Abstracción: a veces no necesitamos saber cómo está hecha una cosa para utilizarla; la abstracción consiste en ocultar los detalles de algo y quedarnos con lo que nos interesa. En programación, la abstracción permite usar un código sin saber nada de él; basta con saber cómo llamar a la función.
4. Legibilidad: dividiendo un programa en funciones se consigue hacer que el código sea más fácil de leer.
5. Mantenimiento: es más fácil actualizar y mantener un programa cuando se ha dividido en funciones: si tenemos que depurar errores o actualizar una parte de código, sólo tenemos que modificar la función que hace las funciones que nos interesan.

Las funciones en programación son muy parecidas a las funciones en matemáticas: tenemos que saber cómo llamarlas, qué datos pasarles y recoger los resultados (si los hay). Gráficamente, se puede representar de esta forma:



Es decir: la función se puede entender como una máquina a la que pasamos una entrada, la procesa de algún modo (que podemos o no conocer) y devuelve un resultado.

Cómo se define una función en Python:

Se usa la palabra reservada `def`, a continuación se pone el nombre de la función (en minúsculas) y unos paréntesis; dentro de los paréntesis se pone el nombre de las variables que vaya a usar la función en su interior (estas variables desaparecen cuando la función termina su función) y a continuación se ponen dos puntos. Ejemplo:

```
def mi_funcion(x):  
    return x*2
```

La función anterior se llama `mi_funcion`, se le pasa un valor y en su interior la función calcula el doble del valor que se le pasa y lo devuelve usando `return`.

Ojo: en el nombre de una función no se ponen espacios en blanco ni tildes, paréntesis,...

Ejemplo:

#Ejemplo de una función que recibe un número y lo multiplica por dos.

#Definición de funciones:

```
def duplicadora(x):  
    return x*2
```

```
num = int(input("Dame un número: "))  
doble_num = duplicadora(num)  
print(f"El doble de {num} es: {doble_num}")
```

```
Dame un número: 67  
El doble de 67 es: 134
```

Un par de detalles importantes:

- El nombre de la función debería ser significativo: el nombre debería indicar para qué sirve.
- No es buena idea usar como nombre de la función el de otra función que ya existe.
- Una función puede usarse a partir de que se ha definido en el código del programa, por eso es buena idea definir funciones que se vayan a usar al comienzo del programa.

Una función puede tener un parámetro, varios o ninguno:

```
#Ejemplo de funciones que reciben distinto número de parámetros.

#Definición de funciones:
def duplicadora(x):
    return x*2

def divisor(x,y):
    if y != 0:
        return x/y
    else: #No se puede dividir por 0
        return -1

def muestra_info():
    print("Este es un programa que sirve para probar funciones en Python")
    print("Esperamos que les guste!")

muestra_info()
num = int(input("Dame un número: "))
doble_num = duplicadora(num)
print(f"El doble de {num} es: {doble_num}")
numerador = int(input("Dame un numerador (el de arriba): "))
denominador = int(input("Dame un denominador (el de abajo): "))
resultado = divisor(numerador,denominador)

if resultado == -1:
    print("No se pudo hacer la división porque el denominador no puede ser 0")
else:
    print(f"{numerador} / {denominador} = {resultado}")
```

Y al ejecutarlo:

```
===== RESTART: C:\Python\Clase\funciones\funciones2_ejemplos.py
Este es un programa que sirve para probar funciones en Python
Esperamos que les guste!
Dame un número: 6
El doble de 6 es: 12
Dame un numerador (el de arriba): 77
Dame un denominador (el de abajo): 8
77 / 8 = 9.625
```

En el código anterior se puede ver una función que recibe un parámetro; otra función que recibe dos parámetros y otra que no recibe ninguno.

Parámetros opcionales: En Python existen funciones que admiten parámetros opcionales, es decir: si al llamar a la función se usa un parámetro en la llamada, entonces se emplea, dentro de la función, ese valor; en caso contrario se emplea el valor por defecto que se indica al definir la función.

Ejemplo:

```
def f(x=2):
    return x**x

print(f())
print(f(4))
```

```
=====
4
256
>>>
```

Cómo devolver más de un valor en una función:

Es posible que en una función queramos devolver más de un valor; se puede hacer indicando, a continuación del return, los valores a devolver separados por comas.

La forma de recoger los valores devueltos por la función será usando las variables necesarias, separadas por comas.

#Ejemplo de funcion que devuelve varios valores.

```
def obtener_coordenadas():
    latitud = 40.7128
    longitud = -74.0060
    return latitud, longitud

# Llamar a la función y asignar los valores devueltos a dos variables
latitud, longitud = obtener_coordenadas()

print(f"Latitud: {latitud}")
print(f"Longitud: {longitud}")
```

Y al ejecutarlo:

```
Latitud: 40.7128
Longitud: -74.006
```

A continuación se muestra un ejemplo en el que se devuelven tres valores distintos:

```
#Ejemplo de funcion que devuelve varios valores.

def valores_estelares():
    ascension = "40 horas, 20 minutos, 5 segundos"
    long = "18°, 5 minutos, 19 segundos"
    mag = -1
    return ascension, long, mag

# Llamar a la función y asignar los valores devueltos a tres variables
ascension_recta, longitud, magnitud = valores_estelares()

print(f"Ascensión recta: {ascension_recta}")
print(f"Longitud: {longitud}")
print(f"Magnitud: {magnitud}")
```

Daríamos este resultado al ejecutarlo:

```
Ascensión recta: 40 horas, 20 minutos, 5 segundos
Longitud: 18°, 5 minutos, 19 segundos
Magnitud: -1
```




Cómo agrupar funciones en Python:

Cuando se han definido unas cuantas funciones en Python se pueden agrupar en un fichero con la extensión “.py”; después de haber creado el archivo .py hay que importarlo en el programa para poder usar las funciones que contiene. Ejemplo:

```
funciones_pruebamodulo.py - C:/Python/Clase/funciones/funciones_pruebamodulo.py (3.13.0)
File Edit Format Run Options Window Help
1 #Ejemplo para probar la importación de un módulo en un programa:
2
3 import mis_funciones
4
5 print(mis_funciones.saludar("Juan"))
6 print(mis_funciones.sumar(3, 4))

mis_funciones.py - C:/Python/Clase/funciones/mis_funciones.py (3.13.0)
File Edit Format Run Options Window Help
1 # mis_funciones.py
2 def saludar(nombre): return f"Hola, {nombre}!"
3 def sumar(a, b): return a + b
4 |
```

Los módulos se agrupan en paquetes (en otros lenguajes de programación se llaman librerías o bibliotecas) dentro de un directorio; dentro del directorio hay que poner un fichero vacío que se llame `__init__.py` (ojo: son dos guiones bajos a cada lado del init)

Disco local (C:) > Python > Clase > funciones > textos				Buscar en textos
Nombre	Fecha de modificación	Tipo	Tamaño	
 <code>__init__.py</code>	20/01/2025 20:44	Python File	0 KB	
 <code>textos_1.py</code>	20/01/2025 20:30	Python File	1 KB	
 <code>textos_2.py</code>	20/01/2025 19:50	Python File	1 KB	

Para hacer la importación en el programa se selecciona el módulo o los módulos que queremos usar:

```
#Ejemplo de importación de módulos de un paquete o librería o biblioteca

from textos import textos_1, textos_2
```

Aquí surge un problema: ¿cómo sabe Python en dónde está el directorio? Dos posibles soluciones:

1: Podemos añadir, en el propio programa, la ruta del directorio donde se guardan los módulos: se pueden usar rutas absolutas o relativas, pero tiene más sentido usar rutas relativas si el paquete se entrega junto con el resto de archivos para su instalación.

#Ejemplo de importación de módulos de un paquete o librería o biblioteca
#indicamos a Python dónde está el directorio

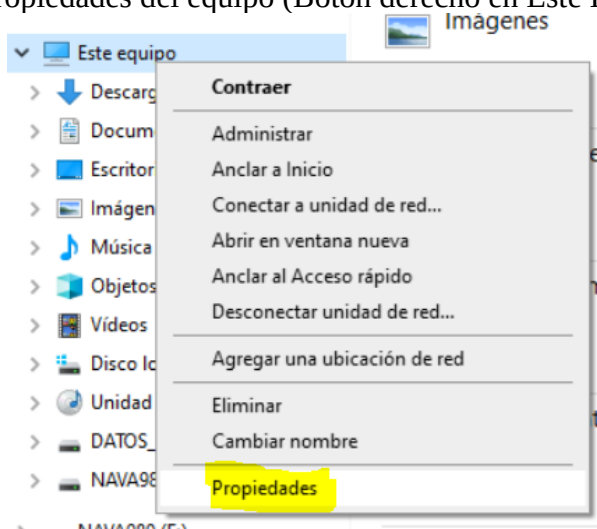
```
import sys
sys.path.append('..\mis_librerías')

from textos import textos_1, textos_2
```

2: Añadir el directorio al PYTHONPATH

Dependerá del S.O. que estemos usando:

En Windows: Desde las propiedades del equipo (Botón derecho en Este Equipo):



A continuación: Configuración avanzada del sistema:

[Lee los Términos de licencia del software de Microsoft](#)

Opciones de configuración relacionadas

[Configuración de BitLocker](#)

[Administrador de dispositivos](#)

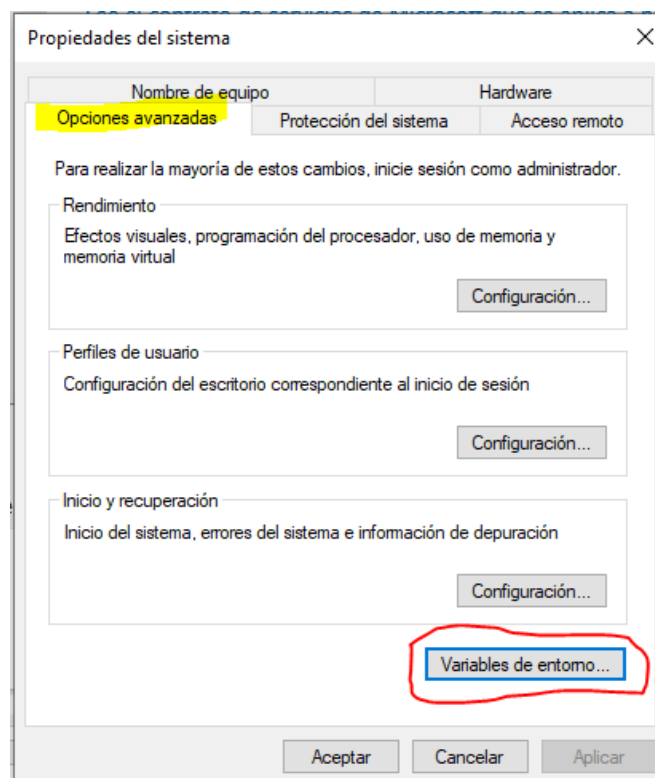
[Escritorio remoto](#)

[Protección del sistema](#)

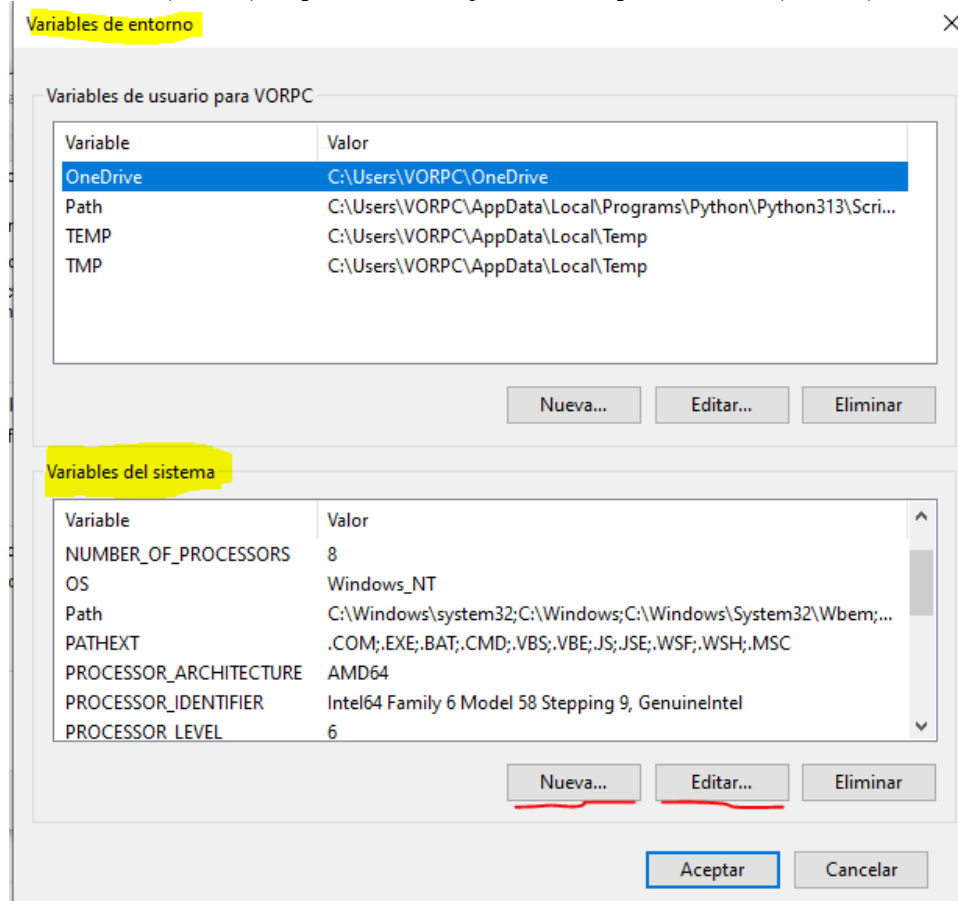
[Configuración avanzada del sistema](#)

[Cambiar el nombre de este equipo \(avanzado\)](#)

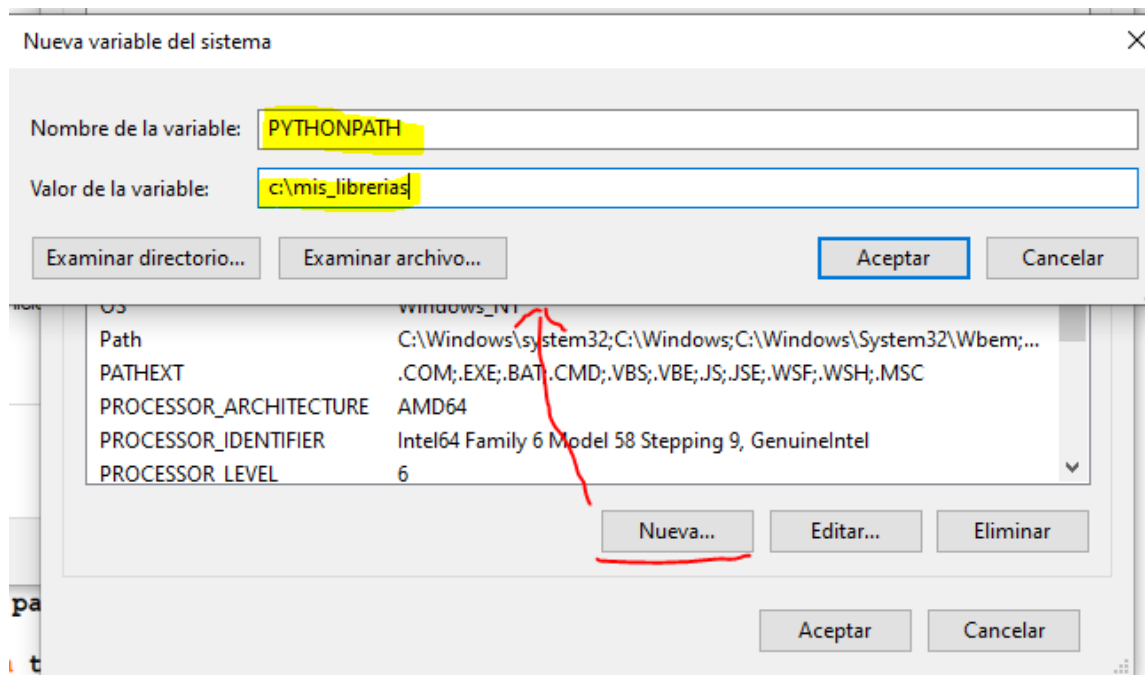
En la solapa de Opciones Avanzadas: Variables de Entorno:



Desde aquí pueden pasar dos cosas: que ya exista la variable de Sistema PYTHONPATH en ese caso, la editamos (Editar) o que no exista y tenemos que añadirla (Nueva):



Para añadirla:



En Unix/Linux/macOS:

Desde un terminal abierto como root (o usando sudo):

Editamos el archivo .bashrc o .bash_profile y añadimos estas líneas:

```
export PYTHONPATH=$PYTHONPATH:/mis_librerias
```

guardamos el archivo y ejecutamos `source ~/.bashrc`

4. MANEJO DE EXCEPCIONES

Las excepciones son errores que se producen durante la ejecución y no puede evitarse que ocurran porque muchas veces dependen de las acciones del usuario. Para evitar que se produzcan los errores de ejecución se puede intentar capturar los posibles errores y asociar unas líneas de código que se ejecutarán en caso de que se produzca un error de ejecución.

Ejemplo típico:

```
#Cómo provocar un error ejecución

a = int(input("Dame un número: "))
b = int(input("Dame otro número: "))

print(a/b)
```

```
Dame un número: 45
Dame otro número: 0
Traceback (most recent call last):
  File "C:/Python/Clase/funciones/funcion_opcional.py",
    print(a/b)
ZeroDivisionError: division by zero
```

Como no podemos evitar que el usuario meta el valor que quiera, tenemos que controlar los posibles errores y para ello se usan las palabras reservadas *try* y *except*. *Try* se usa para contener las líneas en las que pueden ocurrir el error y *except* contiene las líneas que se ejecutarán solo en el caso de que ocurra el error.

A continuación se muestra un ejemplo de cómo controlar un error de división por cero:

```
#Cómo controlar un error usando try except.
```

```
a = int(input("Dame un número: "))
b = int(input("Dame otro número: "))

try:
    print(a/b)
except ZeroDivisionError:
    print("El segundo número no puede ser 0")
    exit()
```

```
>>>
===== RESTART: C:/Python/Clas
Dame un número: 45
Dame otro número: 0
El segundo número no puede ser 0
```

Hay que tener en cuenta que, como se ve en el ejemplo, hay que especificar el error que queremos controlar; en este caso es la división por cero; si se produjera otro error distinto no lo estaríamos controlando.

En el ejemplo siguiente vemos que se produce un error distinto del que habíamos previsto:

```
#Cómo controlar un error usando try except.
```

```
a = int(input("Dame un número: "))
b = int(input("Dame otro número: "))

try:
    print(a/b)
except ZeroDivisionError:
    print("El segundo número no puede ser 0")
    exit()
```

```
===== RESTART: C:/Python/Clase/fu
Dame un número: uno
Traceback (most recent call last):
  File "C:/Python/Clase/funciones/funcio
    a = int(input("Dame un número: "))
ValueError: invalid literal for int() wi
```


Con *except* podemos controlar varios errores distintos, pero tenemos que incluir dentro del *try* las líneas en las que pueden ocurrir esos errores.

#Cómo controlar un error usando try except.

```
try:
    a = int(input("Dame un número: "))
    b = int(input("Dame otro número: "))
    print(a/b)
except (ZeroDivisionError, ValueError):
    print("Entrada incorrecta: asegúrese de meter sólo números. El segundo no puede ser 0.")
    exit()
```

Dame un número: trece

Entrada incorrecta: asegúrese de meter sólo números. El segundo no puede ser 0.

Dos aspectos más que deben tenerse en cuenta en el tratamiento de excepciones: si indicamos más de una condición de error en *except* tenemos que ponerlas entre paréntesis y separadas por comas (como se ve en el ejemplo). Algo no que debemos hacer es definir variables dentro del *try* porque esas variables si la excepción (el error) ocurre antes de llegar a la línea donde se define la variable, entonces no se habrá definido la variable.

Puede encontrarse un listado de las excepciones que existen en la siguiente URL:

https://www.tutorialspoint.com/python/standard_exceptions.htm

5. CONTENEDORES (CONTAINERS)

En Python un contenedor es un tipo de objeto que pueden contener y manejar otros objetos, por ejemplo, una lista puede contener una colección de datos (de cualquier tipo: números enteros, caracteres,...).

Características de los contenedores

- Pueden ser inmutables (no pueden cambiarse sus elementos) o no (sus elementos pueden cambiarse).
- Ordenados o no ordenados.
- Iterables (se pueden recorrer usando bucles).

Tipos de contenedores y sus características (todos son iterables):

- Listas: ordenadas y mutables (cambiables). Admite distintos tipos de datos.
- Tuplas: ordenadas e inmutables (no pueden cambiarse).
- Conjuntos: no ordenados e inmutables.
- Diccionarios: no ordenados; las claves son inmutables y únicas.

5.1 Listas:

Una lista es un contenedor de objetos, que pueden ser de distintos tipos; los objetos tienen un orden. Se pueden definir de dos formas posibles: con paréntesis o con llaves. Si la lista se define con valores iniciales, entonces se usan corchetes.

#Distintas formas de definir listas:

```
cosas_varias = ["Pentium", 1.99, 'y', 677]
mas_cosas = list()
frutas = []
```

Pueden añadirse nuevos valores al final de una lista usando `append("Nuevo Valor")`:

#Añadimos valores al final de la lista

```
cosas_varias.append("Calendario")
mas_cosas.append("Carpeta")
mas_cosas.append("Tippex")
```

Puede imprimirse una lista entera usando `print(nombre_lista)`:

#Ejemplo de uso de listas.

#Distintas formas de definir listas:

```
cosas_varias = ["Pentium", 1.99, 'y', 677]
mas_cosas = list()
frutas = []
```

```
print(cosas_varias)
#Añadimos valores al final de la lista
cosas_varias.append("Calendario")
print(cosas_varias)
mas_cosas.append("Carpeta")
mas_cosas.append("Tippex")
print(mas_cosas)
```

Al ejecutar el anterior programa:

```
['Pentium', 1.99, 'y', 677]
['Pentium', 1.99, 'y', 677, 'Calendario']
['Carpeta', 'Tippex']
```

Las listas pueden recorrerse usando bucles:

```
print(cosas_varias)
for i in range(0, 3):
    print(cosas_varias[i])

#Otra forma de recorrer una lista:
for cosa in cosas_varias:
    print(cosa)
```

```
['Pentium', 1.99, 'y', 677, 'Calendario']
Pentium
1.99
y
Pentium
1.99
y
677
Calendario
```

Se puede acceder a cualquier elemento de una lista (únicamente a ese) usando el índice de su posición, teniendo en cuenta que los índices empiezan a contar por el número cero.

```
cosa1 = cosas_varias[0]
print(cosa1)
```

Puede modificarse un valor de una lista mediante el índice del valor:

```
cosas_varias[0] = "AMD"
print(cosas_varias)
['AMD', 1.99, 'y', 677, 'Calendario']
```

También puede eliminarse el último elemento de una lista usando *pop()*:

```
print(cosas_varias)
cosas_varias.pop()
print(cosas_varias)
['Pentium', 1.99, 'y', 677, 'Calendario']
['Pentium', 1.99, 'y', 677]
```

Ojo: si intentamos usar *pop* en una lista vacía se producirá una excepción (error de ejecución).

Pueden combinarse dos listas usando el operador de suma: "+":

```
lakers = ["Magic", "Abdul-Jabbar", "Worthy", "Cooper", "Rambis"]
celtics = ["Bird", "Ainge", "Parish", "McHale", "Johnson"]
celtlakers = lakers + celtics
['Magic', 'Abdul-Jabbar', 'Worthy', 'Cooper', 'Rambis', 'Bird', 'Ainge', 'Parish', 'McHale', 'Johnson']
```

Comprobar si un elemento pertenece o no a una lista puede hacerse usando "elemento" in lista o "elemento" not in lista:

```
print("Romerito" in lakers)
if "Romay" not in lakers:
    print("Romay no jugaba en los Lakers")
False
Romay no jugaba en los Lakers
```

Longitud de una lista (o número de elementos): puede saberse usando *len()*

```
print(f"Número de jugadores del equipo ficticio celtlakers {len(celtlakers)}")
for i in range(0, len(lakers)):
    print(f"Elemento {i}: {lakers[i]}")
Número de jugadores del equipo ficticio celtlakers 10
Elemento 0: Magic
Elemento 1: Abdul-Jabbar
Elemento 2: Worthy
Elemento 3: Cooper
Elemento 4: Rambis
```

Resumen de métodos y funciones relacionados con listas:

Aclaración:

- los métodos se llaman poniendo un punto y el nombre del método a continuación:
`lista_coches.append(item)`
- las funciones se usan poniendo entre paréntesis el nombre de la lista a la que aplicamos la función; además al llamar a la función es habitual recoger lo que la función nos devuelve.

MÉTODOS DE LAS LISTAS	
<code>Lista.append()</code>	Añade un elemento al final de la lista.
<code>Lista.extend()</code>	Añade los elementos de una lista (o iterable) al final de la lista actual.
<code>Lista.insert()</code>	Inserta un elemento en una posición específica.
<code>Lista.remove()</code>	Elimina la primera aparición del elemento especificado.
<code>Lista.pop([posición])</code>	Elimina y devuelve el elemento en la posición indicada (por defecto, el último elemento).
<code>Lista.clear()</code>	Elimina todos los elementos de una lista.
<code>Lista.index(ítem, [inicio], [fin])</code>	Devuelve el índice del primer elemento con el valor especificado.
<code>Lista.count()</code>	Devuelve el número de veces que un elemento aparece en la lista.
<code>Lista.sort(key=None, reverse=False)</code>	Ordena los elementos de una lista en su lugar (opcionalmente se puede usar una función clave y/o un valor booleano de reverso).
<code>Lista.reverse()</code>	Invierte los elementos de una lista.
<code>Lista.copy()</code>	Devuelve una copia superficial de la lista.

FUNCIONES DE LAS LISTAS	
<code>len(lista)</code>	Devuelve el número de elementos de la lista (su longitud).
<code>max(lista)</code>	Devuelve el elemento con el valor máximo de la lista.
<code>min(lista)</code>	Devuelve el elemento con el valor mínimo de la lista.
<code>sum(lista)</code>	Devuelve la suma de los elementos de la lista (tiene que ser números).
<code>sorted(lista, key=None, reverse=False)</code>	Devuelve una lista nueva ordenada de los elementos de lista, usando una función clave y/o un valor booleano de reverso).
<code>list(iterable)</code>	Convierte un iterable en una lista.

Para ejemplos de estos métodos y funciones, consultar el programa:

`listas_funciones_metodos.py`

5.2 Tuplas:

Una tupa es un contenedor que almacena objetos en un orden específico. Al contrario que las listas, las tuplas no pueden modificarse: una vez creadas con unos determinados valores no pueden modificarse los valores que contiene, ni añadir nuevos elementos ni eliminarlos. Cualquier intento de hacer alguna de estas operaciones provoca una excepción.

Están pensadas para usar datos que no van a poder modificarse dentro del programa; usando tuplas se protegen los valores almacenados para que no puedan ser modificados accidentalmente.

- Posibles ejemplos de uso:
- Coordenadas geográficas
- Fechas que no pueden cambiar
- Constantes (físicas, matemáticas, de ingeniería,...)
- Puntos en espacios de varias dimensiones.
- Almacenar conjuntos de constantes que no van a cambiar. Ej.: direcciones = ('Norte', 'Sur', 'Este', 'Oeste')

Una tupla se define usando paréntesis y comas para separar los valores de la tupla; si la tupla tiene un único elemento hay que poner una coma después de este único valor.

Formas de definir una tupla:

```
#Dos formas de definir una tupla:
tupla1 = ()
tupla2 = tuple()

#Ahora damos valores a la tupla
tupla1 = (1, 2, 3)
tupla2 = ('Madrid', 'Zaragoza', 'Salamanca')|
```

Si hemos dicho que, una vez definida, no puede modificarse una tupla; ¿funcionará lo que se ha hecho en el código anterior? ¿qué está ocurriendo?

Los valores de una tupla no tienen que ser del mismo tipo:

```
cantante1 = ('Michael Jackson', "29-08-1958", 'Gary, Indiana',
            "25/06/2009", "Los Angeles")
```

Se puede imprimir una tupla de varias formas:

```
print(tupla2)
('Madrid', 'Zaragoza', 'Salamanca')
print(cantante1)
('Michael Jackson', '29-08-1958', 'Gary, Indiana', '25/06/2009', 'Los Angeles')
```

```

tupla2 = ('Madrid', 'Zaragoza', 'Salamanca')
for i in range(0,3):
    print(tupla2[i])
Madrid
Zaragoza
Salamanca

```

Una operación curiosa que puede hacerse con las tuplas es desempaquetarlas (asignar sus valores a distintas variables):

```

nombre, fecha_nacimiento, lugar_nacimiento, fecha_fallecimiento, lugar_fallecimiento = cantante1

```

Que es equivalente a hacer:

```

nombre = cantante1[0]
fecha_nacimiento = cantante1[1]
lugar_nacimiento = cantante1[2]
fecha_fallecimiento = cantante1[3]
lugar_fallecimiento = cantante1[4]

```

Respecto a la modificación de tuplas:

Una vez creadas no pueden modificarse (salvo en algún caso excepcional), por lo tanto, lo normal es crear la tupla con los valores desde el principio.

Esto es lo que ocurre cuando intentamos modificar una tupla:

```

tupla_lista = (1, [2,3], 89, 94)
tupla_lista[0] = 777

```

Resultado:

```

Traceback (most recent call last):
  File "D:/Curso24-25/INS/Python/pythonadas/tuplas_modificacion
<module>
    tupla_lista[0] = 777
TypeError: 'tuple' object does not support item assignment

```

Un caso en el que sí se puede modificar el contenido de una tupla es cuando el dato a modificar está dentro de otro tipo de contenedor, por ejemplo: una lista (y se supone que estamos modificando la lista, no la tupla).

```

tupla_lista = (1, [2,3], 89, 94)
tupla_lista[1][0] = 999
print(tupla_lista)

```

Y este es el resultado:

```

(1, [999, 3], 89, 94)

```

Otras operaciones que pueden hacerse con las tuplas son:

Acceso a elementos usando el índice (que puede tener valores positivos o negativos):

```
print(tupla_lista) → (1, [999, 3], 89, 94)
print(tupla_lista[0]) → 1
print(tupla_lista[1]) → [999, 3]
print(tupla_lista[2]) → 89
print(tupla_lista[-1]) → 94
print(tupla_lista[-3]) → [999, 3]
```

Y, en el caso especial en que queramos acceder a un elemento de una lista dentro de una tupla, usamos dos índices: uno para indicar el elemento de la tupla y otro para el elemento dentro de lista. En el ejemplo se muestran los elementos 0 y 1 de la lista que está en segunda posición de la tupla:

```
tupla_lista = (1, [2,3], 89, 94) >>>
tupla_lista[1][0] = 999
print(tupla_lista) → (1, [999, 3], 89, 94)
print(tupla_lista[1][0]) → 999
print(tupla_lista[1][1]) → 3
```

Pueden concatenarse varias tuplas o repetirse: usando el símbolo “+” para su suma o “*” seguido de un número para su repetición:

```
#Operaciones con tuplas:
numeros1 = (88, 9, 100)
numeros2 = (5, 6, 65)
masnumeros = numeros1 + numeros2
triplesnumeros = numeros1 * 3
print(f"Valores de la tupla numeros1: {numeros1}")
print(f"Valores de la tupla numeros2: {numeros2}")
print(f"Concatenación de las tuplas numeros1 + numeros2: {masnumeros}")
print(f"Repetición de la tupla numeros1 (numeros1*3) = {triplesnumeros}")
```

Resultado al ejecutar el programa:

```
Valores de la tupla numeros1: (88, 9, 100)
Valores de la tupla numeros2: (5, 6, 65)
Concatenación de las tuplas numeros1 + numeros2: (88, 9, 100, 5, 6, 65)
Repetición de la tupla numeros1 (numeros1*3) = (88, 9, 100, 88, 9, 100, 88, 9, 100)
```

Otra operación que puede hacerse con tuplas es obtener una subtupla especificando un rango de índices: `tupla[n:m]` -> devuelve los valores de las posiciones n a m-1 de la tupla:

```
tuplagrande = (4, 55, 67, 88, 99, 0, 232, 55, 6, 777, 7, 3454, 1223, 6, 77)
minitupla = tuplagrande[2:6]
print(minitupla)

(67, 88, 99, 0)
```

A continuación se indican métodos y funciones de tuplas:

MÉTODOS DE LAS TUPLAS	
<code>tupla.index(valor, [inicio], [fin])</code>	Devuelve el índice de la primera aparición del valor en la lista; puede especificarse un rango en el que busque.
<code>tupla.count(valor)</code>	Devuelve el número de veces que un valor aparece en la lista.

FUNCIONES DE LAS TUPLAS	
<code>len(tupla)</code>	Devuelve el número de elementos de la tupla.
<code>max(tupla)</code>	Devuelve el elemento con el valor máximo de la tupla.
<code>min(tupla)</code>	Devuelve el elemento con el valor mínimo de la <u>tupla</u> .
<code>sum(tupla)</code>	Devuelve la suma de los elementos de la tupla (tienen que ser números).
<code>sorted(tupla)</code>	Devuelve una lista ordenada de los elementos de la tupla, pero no modifica la tupla.
<code>tuple(iterable)</code>	Convierte un iterable en una <u>tupla</u> .

(ver ejemplos en el programa: `tuplas_funciones_metodos.py`)

5.3 Diccionarios

Un diccionario es un tipo de contenedor que relaciona una serie de objetos, llamados claves, con otros llamados valores. A la unión entre un clave y valor se llama mapeo; el resultado es un conjunto de pares clave-valor.

Puede accederse a un valor a través de su clave, pero no a la inversa (el valor no nos proporciona la clave).

Los diccionarios pueden modificarse, así que pueden añadirse pares clave-valor. Los diccionarios mantienen el orden en que se crearon; no se ordenan y su principal utilidad la asociación entre claves y valores.

Los diccionarios usan llaves para encerrar sus elementos: `{}`. Se pueden definir los diccionarios de dos formas diferentes:

```
mi_diccionario = dict()
tu_diccionario = {}
```

Y posteriormente podemos añadir los pares clave-valor:

```
mi_diccionario = {"El Quijote": "Miguel de Cervantes", "Niebla": "Miguel de Unamuno",
                  "El Buscón": "Francisco de Quevedo", "El camino": "Miguel Delibes"}

tu_diccionario = {"The River": "Bruce Springsteen", "Thriller": "Michael Jackson",
                  "Days Like This": "Van Morrison"}
```

La forma de acceder a los elementos de un diccionario es a través de sus claves:

```
print(mi_diccionario)
print(tu_diccionario)
```


Los diccionarios pueden modificarse:

- Añadiendo nuevos pares clave-valor:
`tu_diccionario["Brothers in arms"] = "Camarón de la Isla"`
- modificando un valor (no pueden modificarse las claves):
`tu_diccionario["Brothers in arms"] = "Dire Straits"`

Podemos recorrer un diccionario iterando sobre las claves, los valores o ambos:

```
#Recorremos el diccionario usando las claves:
for i in tu_diccionario.keys():
    print(i)

#Recorremos el diccionario usando los valores:
for i in tu_diccionario.values():
    print(i)

#Recorremos para mostrar los pares clave-valor
for clave, valor in tu_diccionario.items():
    print(clave + ":" + valor)
```

Para saber si una clave está en el diccionario podemos usar el operador in:

```
if "Thriller" in tu_diccionario:
    print(tu_diccionario.get("Thriller"))
else:
    print("La clave no está en el diccionario")
```

Puede borrarse un par clave-valor usando del:

```
del tu_diccionario["The River"]
```

MÉTODOS DE LOS DICCIONARIOS	
<code>diccionario.clear()</code>	Elimina todos los elementos del diccionario.
<code>diccionario.copy()</code>	Devuelve una copia superficial del diccionario
<code>diccionario.fromkeys(iterable, valor_por_defecto)</code>	Crea un nuevo diccionario con claves del iterable y un valor por defecto.
<code>diccionario.get(clave, valor_por_defecto)</code>	Devuelve el valor de una clave o un valor por defecto si no existe esa clave.
<code>diccionario.items()</code>	Devuelve una vista de pares clave-valor.
<code>diccionario.keys()</code>	Devuelve una vista de todas las claves.
<code>diccionario.values()</code>	Devuelve una vista de todos los valores.
<code>diccionario.pop(clave, valor_por_defecto)</code>	Elimina una clave y devuelve su valor; lanza error si no existe y no se especifica un valor por defecto.
<code>diccionario.popitem()</code>	Elimina y devuelve el último par clave-valor.
<code>diccionario.setdefault(clave, valor_por_defecto)</code>	Devuelve el valor de una clave y, si no existe, la crea con el valor por defecto.
<code>diccionario.update(otro_diccionario_o_iterable)</code>	Actualiza el diccionario con pares clave-valor de otro diccionario o un iterable.

FUNCIONES DE LOS DICCIONARIOS	
len(diccionario)	Devuelve el número de elementos del diccionario.
dict(clave1=valor1,...)	Crea un diccionario con los pares clave-valor que se indiquen. Las claves tienen que ser indicadores válidos para la función dict().
“clave1” in diccionario	Comprueba si una clave está en el diccionario.
del diccionario[“clave”]	Elimina una clave específica del diccionario.
enumerate(diccionario)	Crea un iterable enumerado; junto con un for se usa para recorrer un diccionario.

(ver ejemplos en el programa: diccionarios_funciones_metodos.py)

6. ARCHIVOS O FICHEROS (QUE ES LO MISMO)

Operaciones que pueden hacerse con los archivos:

- Abrir
- Cerrar
- Leer
- Escribir
- Añadir

El primer paso para usar un archivo es que el archivo exista (o bien que se cree al abrirlo en caso de que no exista).

Un fichero puede abrirse de varios modos:

- Solo lectura.
- Lectura y escritura.
- Escritura (machacaría el contenido en caso de que lo tenga).
- Añadir: la información que metamos se añade al final de la que ya existe.

Normalmente, si el archivo se abre para escribir en él lo normal es que se bloquee para que otro usuario no pueda abrirlo al mismo tiempo.

Los modos de apertura en Python son:

r	Lectura.	Si el archivo no existe se genera el error FileNotFoundError.
w	Escritura.	Si el archivo no existe se crea uno nuevo; si existe se sobrescribe en él.
x	Escritura exclusiva.	Crea un nuevo archivo, si ya existe el archivo (mismo nombre, misma ubicación): se genera el error FileExistsError.
a	Añadir.	Abre el archivo para añadir contenido al final sin borrar en el contenido que ya existía. Si el fichero no existe se crear uno nuevo.
b	Binario.	Para escribir/leer en modo binario (no en modo carácter). Se usa en combinación con otros modos: rb o wb.
t	Modo texto.	Es modo por defecto, para escribir o leer texto.
+	Actualización.	Para leer y escribir en el mismo archivo, se usa con r o w: r+ o w+

Abrir un archivo para lectura:

Se usa la función **open()** con dos parámetros: una cadena con el nombre del archivo (que puede ser la ruta completa) y otra cadena modo de apertura.

La función más básica para leer es **read()**, pero existen algunas variantes:

- ✓ **read()**: lee todo un archivo como si fuera una cadena.
- ✓ **readline()**: lee una sola línea cada vez (se usa dentro de un bucle).
- ✓ **readlines()**: devuelve una lista donde cada elemento es una línea del archivo.

Con **read()** leemos un archivo de una vez y lo almacenamos en una variable que luego mostramos con **print()**:

```
archivo = open("jugadoresNBA.txt", "r")
#leemos todo el contenido del archivo y lo almacenamos en la variable contenido.
contenido = archivo.read()
print(contenido)
archivo.close()
```

Otra forma de hacer lo mismo, pero con la estructura **with**:

```
with open("jugadoresNBA.txt", "r") as archivo:
    contenido = archivo.read()

print(contenido)
```

Con **readline()** iremos leyendo las líneas del archivo una por una:

```
print("### Mostramos el archivo línea a línea ###")
with open("jugadoresNBA.txt", "r") as archivo:
    linea = archivo.readline()
    while linea:
        print(linea.strip()) #strip:elimina \n al final de la línea
        linea = archivo.readline()
```

En el ejemplo anterior se lee una primera línea del fichero, se entra en un bucle que se repetirá mientras la variable **línea** tenga contenido (no esté vacía).

La función **readlines()** lee todas las líneas del archivo y devuelve una lista donde cada elemento es una línea del fichero:

```
print("### Mostramos el archivo línea a línea ###")
with open("jugadoresNBA.txt", "r") as archivo:
    linea = archivo.readline()
    while linea:
        print(linea.strip()) #strip:elimina \n al final de la línea
        linea = archivo.readline()
```

Abrir un archivo para escritura:

Se usa la función **open()** con dos parámetros: una cadena con el nombre del archivo (que puede ser la ruta completa) y otra cadena modo de apertura.

Esta función devuelve un objeto fichero (en otros lenguajes se llama un descriptor) que será la forma en que haremos referencia al fichero para las lecturas, escrituras o cierre.

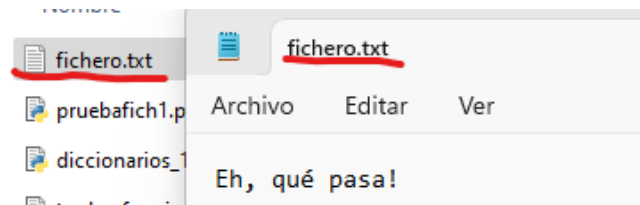
Ejemplo:

```
1 st = open("fichero.txt", "w")
2 st.write("Eh, qué pasa!")
3 st.close()
```

En la línea 1 hemos abierto el archivo “fichero.txt” para escritura. Como el archivo no existe se crea nuevo.

Los parámetros que se pasan en la función open() son el nombre del archivo: “fichero.txt” y el modo de apertura: “w”.

El resultado del programa anterior sería (ojo: no se muestra nada por pantalla):



El fichero se ha creado en el mismo directorio en el que tenemos el fichero con el código.

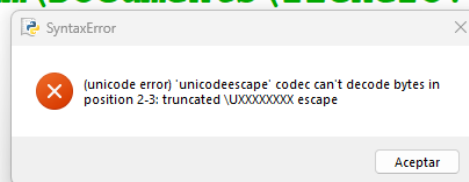
Si queremos hacer referencia a un fichero que no se encuentra en nuestro directorio activo, entonces tendríamos que indicar la ruta (path) completa de esta forma:

```
st = open("C:/Users/fnavam/Documents/fichero.txt", "w")
st.write("Eh, qué pasa!")
st.close()
```

Es importante ver que se ha usado la barra (slash) para separar directorios; es decir: la que se usa en Unix/Linux y no la contrabarra (backslash) que es la que se usa en Windows.

Si nos equivocamos en lo anterior ocurre esto:

```
st = open("C:\Users\fnavam\Documents\fichero.txt", "w")
st.write("Eh, qué pasa!")
st.close()
```



Siempre que abramos un fichero con open() tenemos que cerrarlo, cuando ya no lo necesitamos, usando close() aplicado al objeto fichero, como se ven en la última línea del programa superior.

Usando la estructura **with**:

```
with open("jugadores.txt", "w") as st:
    st.write("Magic Johnson\n")
```

Cuando se llega a la última línea dentro de la estructura with el archivo se cierra.

Si ahora quisiéramos añadir al archivo anterior otro jugador tendríamos que abrir el archivo usando la opción "a":

```
with open("jugadores.txt", "a") as st:
    st.write("Larry Bird\n")
```

En los dos casos anteriores se añadió un carácter de salto de línea ("n") al final del texto que se quería introducir.

Archivos CSV:

Los archivos csv tienen, dentro de cada línea, campos separados por comas; cada líneas se considera una fila o registro y cada campo un campo de una B.D. se usan para intercambio y exportación en BB.DD.

Es necesario importar el módulo csv.

Se muestra un ejemplo a continuación:

```
1 import csv
2
3 # Crear datos estructurados
4 datos = [
5     ["Nombre", "Edad", "Ciudad"],
6     ["Juan", 25, "Madrid"],
7     ["Ana", 30, "Barcelona"],
8     ["Luis", 22, "Valencia"]
9 ]
10
11 # Crear el archivo CSV
12 with open("datos.csv", "w", newline="") as archivo:
13     escritor = csv.writer(archivo)
14     escritor.writerows(datos)
15
16 print("Archivo CSV creado exitosamente.")
```

Se crea una lista con sublistas; cada elemento (o sublista) de la lista datos es una fila.

A continuación abrimos (realmente creamos) el fichero en el que vamos a escribir. Se usa `newline=""` para que no se inserten líneas en blanco extra (ocurre en algunos sistemas operativos). En la línea 13 se ha creado un objeto escritor de la librería csv para escribir líneas en formato CSV. Se escriben varias líneas (en nuestro caso, 4) dentro del archivo datos.csv

Al abrir el archivo creado vemos esto:

Nombre,Edad,Ciudad					
Juan,25,Madrid					
Ana,30,Barcelona					
Luis,22,Valencia					

Es decir: no se ha creado una columna para cada uno de los campos de cada línea.

El problema es que en la configuración de Excel se usa otro separador para los campos de la línea; si usamos punto y coma “;” en lugar de “,” vemos esto:

Nombre	Edad	Ciudad
Juan	25	Madrid
Ana	30	Barcelona
Luis	22	Valencia

Hemos usado, dentro del programa en Python, el delimitador “;”:

```
import csv

# Crear datos estructurados
datos = [
    ["Nombre", "Edad", "Ciudad"],
    ["Juan", 25, "Madrid"],
    ["Ana", 30, "Barcelona"],
    ["Luis", 22, "Valencia"]
]

# Crear el archivo CSV
with open("datos2.csv", "w", newline="") as archivo:
    escritor = csv.writer(archivo, delimiter=';')
    escritor.writerows(datos)

print("Archivo CSV creado exitosamente.")
```

Para leer archivos en formato CSV usamos csv.reader():

```
import csv

with open("datos.csv", newline="") as archivo:
    lector = csv.reader(archivo)
    for fila in lector:
        print(fila)
```

El resultado:

```
['Nombre', 'Edad', 'Ciudad']
['Juan', '25', 'Madrid']
['Ana', '30', 'Barcelona']
['Luis', '22', 'Valencia']
```

Archivos binarios:

En un archivo binario no se almacena texto, sino códigos numéricos o símbolos no interpretables por las persona. Suelen contener datos estructurados (imágenes, archivos, vídeos, ejecutables,...) y requieren un software específico para interpretar los datos que contiene. No son ficheros pensados para ser leídos directamente por personas (al contrario de los de texto).

Ejemplos:

Archivo .dll:

f !f0 !fpN fEñ f0L fEL f'FM FPN fD0 f S f0T f0W fEa fpc fdU f°| f@v f0{ f@y fBz fdU f°| f B f°e fB€ f'e fpN fEñ fàf f0,, fE...
f + f + f ‡ fp‡ fÀ† f § f'S fÅS f0< fB€ fB¶ f °i f “ ” fÄg f~ fp' f · fpN fEñ fpN fEñ f Ò fpÒ f Ö fpÒ fÖð fØ fÀØ
fØÙ fpÙ f Ú fBÚ f Ú fpN fEñ f Ú fpÒ f Ö fpÒ fBŸ fØØ fÀØ fØÙ fpÙ f Ú fBÚ f Ú f Ù f·Ù f·Ù f·Ù f Ý fPB f'B fEB fEB fEB fBß f°
á fpN fEñ fða fpÒ f Ö fpÒ fBŸ fØØ fÀØ fØÙ fpÙ f Ú fBÚ f Ú f Ù f·Ù f·Ù f·Ù f Ý f ä f'B fEB fEB fEB fEB fBß f°á f@a fàä f@a fòæ

Un archivo .exe:

[illegible]

Abrir un archivo binario para escritura y luego para lectura:
(atención a la “b” delante de las comillas dobles.)

```
datos = b"Ejemplo de datos en binario"
with open("archivo.bin", "wb") as fichbin:
    fichbin.write(datos)
```

```
with open("archivo.bin", "rb") as fichbin:
    print(fichbin.read())
```

El resultado:

`b'Ejemplo de datos en binario'`

Realmente no estamos leyendo de un archivo binario (no contiene datos en binario).

Vamos a probar con un archivo realmente binario: abrimos un archivo .exe que hemos copiado a la carpeta donde están los programas de Python:

```
with open ("csi.ni.exe", "rb") as fichbin:
    contenido = fichbin.read()
    print(contenido)
```

En pantalla veremos:

Squeezed text (788 lines).

Significa texto estrujado o texto apretado: el entorno de ejecución de Python nos advierte de que se nos llenará la pantalla de caracteres (y no nos enteraremos de nada). A continuación, vamos a leer solo los 100 primeros bytes del programa y los mostramos por pantalla:

```
with open ("csi.ni.exe", "rb") as fichbin:
    contenido = fichbin.read(100)
    print(contenido)
```

Vemos algo como esto (es una captura parcial):

[illegible]

Como desconocemos la estructura del programa y el significado de los caracteres esos datos no nos dicen nada.

Otra posibilidad que tenemos, para el caso anterior, es usar un función que nos traduzca (hasta cierto punto) lo que leemos del archivo al formato hexadecimal: la función `hexlify`; para ello hay que importar el módulo `binascii`.

A continuación se muestra la lectura de los 100 primeros bytes de un archivo .exe:

```
import binascii
```

```
with open("csi.ni.exe", "rb") as fichbin:
    contenido = fichbin.read(100)
    print(binascii.hexlify(contenido))
```

Y el resultado (captura incompleta):

[illegible]

A continuación vamos a generar un archivo de imagen, en formato .BMP, de 100x100 píxeles que abriremos con un visor de imágenes desde el sistema operativo. Se trata de una captura parcial.

#Generamos un fichero de tipo .bmp, con un tamaño de

```
# Cabecera de un archivo BMP de 24 bits (54 bytes)
```

```
cabecera = b'BM' + (54+100*100*3).to_bytes(4, 'little') + b'\x00\x00\x00\x00'
cabecera += (100).to_bytes(4, 'little') + (100).to_bytes(4, 'little')
cabecera += (100*100*3).to_bytes(4, 'little') + b'\x13\x0B\x00\x00\x00\x00'
```

```
# Datos de la imagen (100x100 píxeles, color rojo)
```

```
datos = b'\x00\x00\xff' * (100 * 100)
```

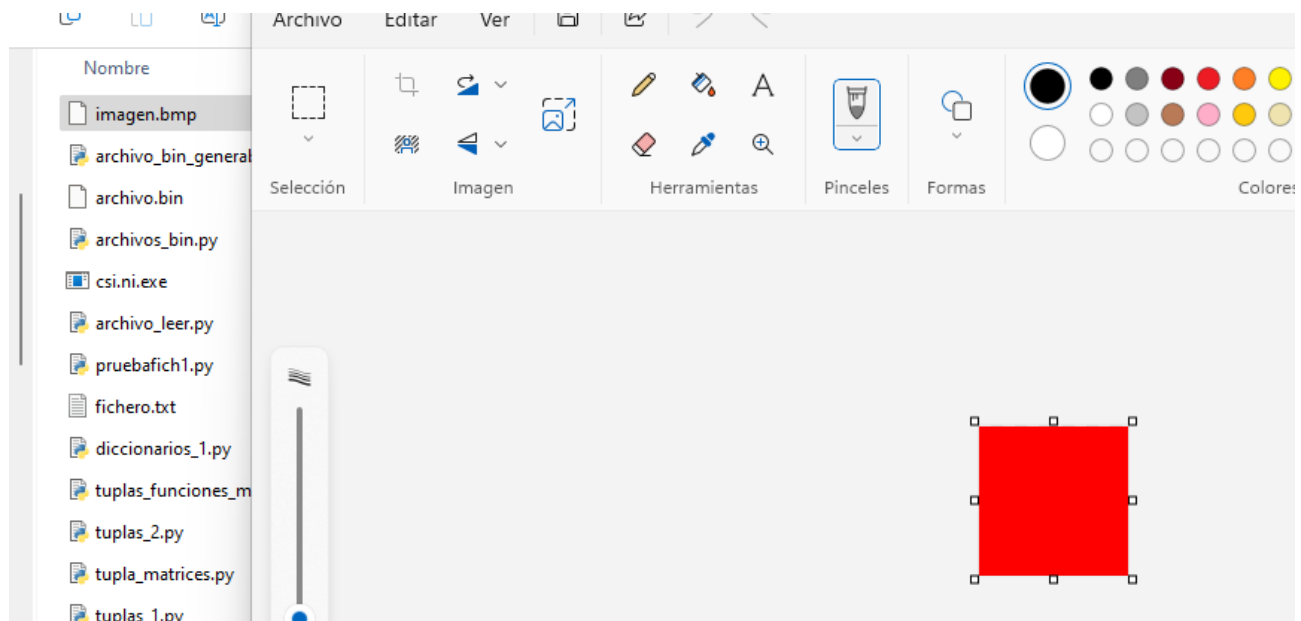
```
# Guardar el archivo
```

```
with open("imagen.bmp", "wb") as archivo:
    archivo.write(cabecera + datos)
```

El formato de almacenamiento es *little endian*, lo que significa que se almacena primero el byte menos significativo; lo que se ve en la variable *datos* es el código hexadecimal del color, pero almacenado a la inversa por este motivo.

Se usa el sistema RGB: 1 byte para cada uno de los tres colores básicos, con valores que van desde 0 a hasta FF; en decimal: 0 a 255.

El resultado, abriendo el archivo con Paint, es este:



Ahora vamos a partir de una imagen que tenemos en nuestro PC, el archivo se llama *milhouse.PNG* y vamos a leer una parte de él, creamos un fichero binario y copiamos parte de lo leído en otro archivo llamado *cacho_milhouse.PNG*

#Abrimos un archivo de imagen y copiamos parte de él en otro fichero.

```
with open("milhouse.PNG", "rb") as fichimg:
    contenido = fichimg.read(30000)
    #print(contenido)

with open("cacho_milhouse.PNG", "wb") as otrofich:
    otrofich.write(contenido)
```

El archivo original tiene un tamaño de 308 KB; vamos a copiar, aproximadamente, una décima parte del fichero original (unos 30 Kbytes).

El resultado, abriendo con Paint el archivo generado por nosotros, es este:



Bibliografía: *The Self-Taught Programmer*. Althoff, Cory. Ed. Robinson. ISBN: 978-1-47214-710-3

Webs:

<https://www.w3schools.com>

<https://www.python.org>

<https://stackoverflow.com>

<https://www.codigopiton.com/como-hacer-switch-case-en-python/>

<https://j2logo.com/>


https://www.tutorialspoint.com/python/standard_exceptions.htm

33 Larry Bird
12 Jerry Sichting
00 Robert Parish
44 Danny Ainge
5 Bill Walton
3 Dennis Johnson
8 Scott Wedman
34 Rick Carlisle
32 Kevin MacHale
50 Greg Kite
11 Sam Vincent
45 David ThirdKill
35 Sly Williams

33 Kareem Abdul-Jabbar
21 Michael Cooper
32 Magic Johnson
1 Earl Jones
25 Mitch Kupchak
12 Ronnie Lester
11 Bob McAdoo
40 Mike McGee
43 Chuck Nevitt
31 Kurt Rambis
4 Byron Scott
35 Larry Spriggs
52 Jamal Wilkes
42 James Worthy
34 Petur Gudmundsson

Por ejemplo:


Python

 Copiar

```
mi_tupla = (1, 2, 3)
mi_tupla[0] = 10 # Esto genera un
```

Sin embargo, **si la tupla contiene un objeto mutable** como una lista, puedes modificar ese objeto dentro de la tupla, aunque no puedas cambiar la estructura de la tupla en sí. Por ejemplo:

Python

 Copiar

```
mi_tupla = (1, [2, 3])
mi_tupla[1][0] = 99 # Esto es vál:
print(mi_tupla) # Salida: (1, [99
```

Números aleatorios:

1. Números enteros aleatorios

`random.randint(a,b)`: genera número entero aleatorio en el rango [a,b] incluyendo ambos extremos.

Ej.:

```
import random

#Generamos 10 números aleatorio entre 1 y 10

print("Ahora generamos 10 números aleatorios:")
for i in range(10):
    print(random.randint(1,10))
```

`random.randrange(start, stop, step)`: genera un número entero aleatorio dentro del rango especificado por start, stop y step.

Es decir: en el ejemplo se generan número aleatorios entre 0 y 99 que sean múltiplos de 3.

Ej.:

```
import random

#Generamos 10 números aleatorio entre 0 y 99, que sean múltiplos de 3
print("Ahora generamos 10 números aleatorios:")
for i in range(10):
    print(random.randrange(0,100,3))
```

2. Números de punto flotante aleatorios:

`random.random()`: genera números aleatorios en el rango [0.0, 1.0) ojo: se excluye el segundo extremo.

Ej.:

```
import random

#Generamos 10 números aleatorios entre 0.0 y 1.0 (este último no incluido)
for i in range(10):
    print(random.random())
```

`random.uniform(a,b)`: genera números aleatorios en el rango [a, b] ahora sí se incluye el segundo extremo.

Ej.:

```
import random
#Generamos 10 números aleatorios entre 5 y 10 (incluidos ambos extremos)
for i in range(10):
    numero_aleatorio = random.uniform(5, 10)
    print(numero_aleatorio)
```

3. Elegir elementos aleatorios de una secuencia:

`random.choice(seq)`: elige un elemento aleatorio de una secuencia (lista, tupla, cadena, etc).

Ej.:

```
import random
#Generamos 5 elementos aleatorios de una lista de elementos
elementos = ["reloj", "libro", "flexo", "cable", "teclado", "ratón", "calendario", "disco"]
for i in range(5):
    print(random.choice(elementos))
```

`random.shuffle(lista)`: mezcla de forma aleatoria los elementos de una lista.

Ej.:

```
import random
#Reordenamos de forma aleatoria los elementos de una lista:
canciones = ["Harry's Game", "Hourglass", "Good Vibrations", "Brian Boru's March", "Outlaws"]
print("Orden inicial de las canciones:")
print(canciones)
print("Orden aleatorio (modo shuffle en un reproductor de cd):")
random.shuffle(canciones)
print(canciones)
```

`random.sample(lista, k)`: devuelve k elementos únicos de la lista (población)

```
# Definir una función que recibe una lista como argumento
def procesar_lista(mi_lista):
    for elemento in mi_lista:
        print(elemento)
```

```
# Crear una lista
mi_lista = [1, 2, 3, 4, 5]
```

```
# Pasar la lista a la función
procesar_lista(mi_lista)
```

```
# Definir una función que crea y devuelve una lista
def crear_lista():
    lista = [1, 2, 3, 4, 5]
    return lista
```

```
# Llamar a la función y almacenar la lista devuelta en una variable
mi_lista = crear_lista()
```

```
print(mi_lista) # Salida: [1, 2, 3, 4, 5]
```

```
# Definir una función que modifique una lista
def modificar_lista(lista):
    lista.append(6)
    lista[0] = 10
```

```
# Crear una lista
mi_lista = [1, 2, 3, 4, 5]
```

```
# Llamar a la función y pasar la lista como parámetro
modificar_lista(mi_lista)
```

```
# Imprimir la lista modificada
print(mi_lista) # Salida: [10, 2, 3, 4, 5, 6]
```

```
Espaciado: indentación = 4 espacios (error) → ejemplo for:
for i in range(100):
    print("Hola mundo")
```

NoneType: siempre tienen el valor None (nada), para representar la ausencia de un valor.

Constantes y variables:

Constante: valor que no cambia durante la ejecución del programa.

Variable: referencia, mediante un nombre, a un valor que puede cambiar durante la ejecución del programa. A las variables se les asigna un valor mediante el operador de asignación: “=”

Incremento/decremento del valor de una variable: $x = x + 1 \rightarrow$ abreviado: $x+=1$

Reglas para nombrar las variables:

1) No espacios

2) Solo se admiten letras, números y el guion bajo “_”

3) No se puede comenzar el nombre de la variable con un número, sí con un guion bajo pero tienen un significado especial.

Sintaxis: las reglas y principios que tiene que seguir la estructura de las sentencias en un lenguaje dado.

Errores y excepciones: si no se cumple la sintaxis de Python aparecerán errores en el programa.

No se pueden ejecutar programas con errores de sintaxis.

Python informa del fichero y la línea del error y nos dice qué tipo de error es.

Existen dos tipos de errores: errores de sintaxis y excepciones. Las excepciones tienen que tratarse desde el código del programa.

Operadores aritméticos: tabla de operadores aritméticos.

Operadores de comparación.

Operadores lógicos.

Cualquier programa puede hacerse con solo tres tipos de estructuras:

Statement: partes de un lenguaje de programación (comandos, cálculos)

- lineal

- condicional (if-then-else)

- repetitivas/bucles (for, while).

Conditional statements: condicionales. Estructuras de control para decidir, después de evaluar una condición, si se ejecuta un conjunto de líneas u otras (o ninguna).

Bucles:

Bucles for

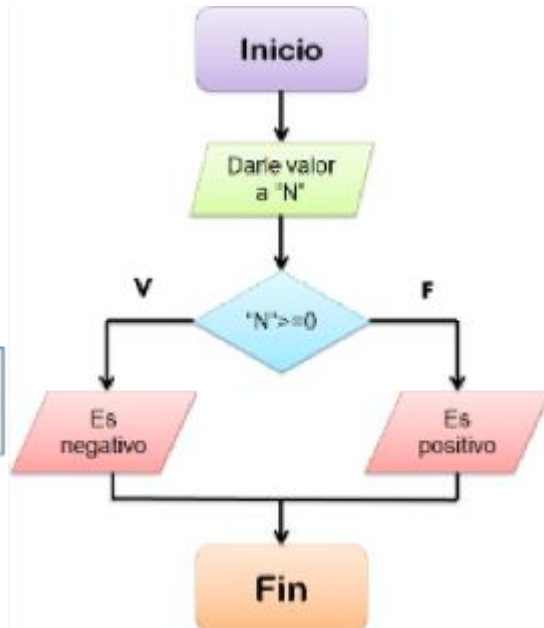
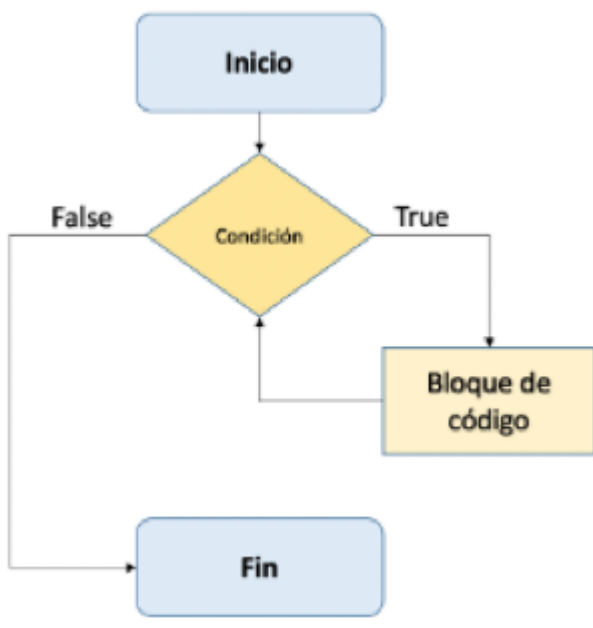
Bucles While

Break

Continue

Bucles recorriendo conjunto o diccionario

Scope de las variables.



Crear un Módulo

1. **Define tus funciones en un archivo:** Crea un archivo con extensión .py y define tus funciones allí. Por ejemplo, crea un archivo llamado mis_funciones.py:

```
python
# mis_funciones.py
def saludar(nombre):
    return f"Hola, {nombre}!"
```

```
def sumar(a, b):
    return a + b
```

2. **Importa el módulo en tus programas:** En tus otros programas, puedes importar y usar las funciones definidas en mis_funciones.py:

```
python
# programa1.py
import mis_funciones
```

```
print(mis_funciones.saludar("Juan"))
print(mis_funciones.sumar(3, 4))
```

Crear un Paquete

1. **Organiza tus módulos en una carpeta:** Crea una carpeta y coloca tus archivos .py dentro de ella. Asegúrate de incluir un archivo vacío llamado __init__.py en la carpeta para que Python la reconozca como un paquete. Por ejemplo:
2. mi_paquete/
3. __init__.py
4. mis_funciones.py
5. otras_funciones.py
6. **Define tus funciones en los archivos:** Define tus funciones en los archivos .py dentro de la carpeta mi_paquete.
7. **Importa el paquete en tus programas:** En tus otros programas, puedes importar y usar las funciones definidas en el paquete:

```
python
# programa2.py
from mi_paquete import mis_funciones, otras_funciones
```

```
print(mis_funciones.saludar("Ana"))  
print(otras_funciones.otrafuncion())
```


Esto te permitirá organizar y reutilizar tus funciones de manera eficiente en múltiples programas.

Espero que esto te sea útil. ¿Hay algo más en lo que pueda ayudarte? 😊

Enviar un mensaje a Copilot

¡Claro! Aquí tienes un ejemplo de cómo usar el método `sort()` con el parámetro `reverse=True` en Python:


Python

 Copiar

```
lista = [3, 1, 4, 2]  
lista.sort(reverse=True)  
print(lista)
```

Resultado:

Python


 Copiar

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

En este ejemplo, la lista `[3, 1, 4, 2]` se ordena en orden descendente debido al uso del parámetro `reverse=True`.

Por supuesto, aquí tienes un ejemplo de cómo usar el método `sort()` con el parámetro `key` en Python:

Python


 Copiar

```
# Lista de tuplas con nombre y edad
personas = [("Ana", 25), ("Juan", 20),

# Ordenar la lista de tuplas por la se
personas.sort(key=lambda x: x[1])
print(personas)
```

Resultado:

Python


 Copiar

```
[('Juan', 20), ('Ana', 25), ('Carlos',
```

En este ejemplo, la lista de tuplas `personas` se ordena en función de la edad (segunda posición de cada tupla) utilizando una función `lambda` como clave (`key`).

¡Entendido! Aquí tienes un ejemplo de cómo usar el método `sort()` con el parámetro `key` en una lista sencilla:

Python


 Copiar

```
# Lista de palabras
palabras = ["banana", "manzana", "cereza"]

# Ordenar la lista de palabras por la longitud
palabras.sort(key=len)
print(palabras)
```

Resultado:

Python

 Copiar

```
['durazno', 'banana', 'manzana', 'cereza']
```


En este ejemplo, la lista `palabras` se ordena en función de la longitud de cada palabra utilizando la función `len` como clave (`key`).

Espero que este ejemplo te sea de ayuda. Si tienes alguna otra pregunta o necesitas más ejemplos, estaré encantado de ayudarte.

El parámetro `key` en el método `sort()` y en la función `sorted()` de Python puede ser cualquier función que tome un solo argumento y devuelva un valor que se utilice para la comparación. Aquí tienes algunos ejemplos comunes de valores que se pueden usar para `key` :

1. `str.lower` : Para ordenar una lista de cadenas ignorando las diferencias entre mayúsculas y minúsculas.


Python

 Copiar

```
palabras = ["Banana", "manzana", "ca  
palabras.sort(key=str.lower)  
print(palabras) # Resultado: ['Bar
```

2. `len` : Para ordenar una lista de cadenas por la longitud de las cadenas.


Python

 Copiar

```
palabras = ["banana", "manzana", "ca  
palabras.sort(key=len)  
print(palabras) # Resultado: ['du
```

3. **Funciones** `lambda` : Para ordenar según un criterio personalizado.


Python

 Copiar

```
# Ordenar por la última letra de ca  
palabras = ["banana", "manzana", "ca  
palabras.sort(key=lambda x: x[-1])  
print(palabras) # Resultado: ['bar
```

4. **Atributos de objetos:** Para ordenar una lista de objetos según un atributo específico.

Python

 Copiar


```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

personas = [Persona("Ana", 25), Persona("Carlos", 30)]
personas.sort(key=lambda x: x.edad)
for persona in personas:
    print(persona.nombre, persona.edad)
```

El parámetro `key` puede ser cualquier función que convierta los elementos de la lista en valores que se puedan comparar directamente. Esto proporciona una gran flexibilidad para ordenar listas según diferentes criterios.

Una "copia superficial" en Python, también conocida como "shallow copy", significa que la copia contiene referencias a los mismos objetos que los elementos originales en la lista. Esto es útil para duplicar la estructura de la lista, pero no los objetos en sí. Aquí tienes un ejemplo para entenderlo mejor:

Python

 Copiar

```
import copy

# Lista original con una lista anidada
original = [[1, 2, 3], [4, 5, 6]]

# Copia superficial de la lista original
copia_superficial = original.copy()

# Modificamos un elemento en la copia
copia_superficial[0][0] = 99

print("Original:", original)
print("Copia superficial:", copia_superficial)
```

Resultado:

Python

 Copiar

```
Original: [[99, 2, 3], [4, 5, 6]]
Copia superficial: [[99, 2, 3], [4, 5, 6]]
```

En este ejemplo, modificar un elemento dentro de la copia superficial también cambia ese elemento en la lista original. Esto ocurre porque la copia superficial solo copia las referencias a los objetos en la lista, no los objetos en sí.

Si necesitas una copia más "profunda" que copie los objetos anidados también, debes usar una "copia profunda" ("deep copy"):

Python

 Copiar

```
import copy

# Lista original con una lista anidada
original = [[1, 2, 3], [4, 5, 6]]


# Copia profunda de la lista original
copia_profunda = copy.deepcopy(original)

# Modificamos un elemento en la copia
copia_profunda[0][0] = 99

print("Original:", original)
print("Copia profunda:", copia_profunda)
```

Resultado:

Python

 Copiar

```
Original: [[1, 2, 3], [4, 5, 6]]
Copia profunda: [[99, 2, 3], [4, 5, 6]]
```

En este caso, la lista original permanece sin cambios porque la copia profunda crea nuevos objetos en lugar de solo copiar referencias.

