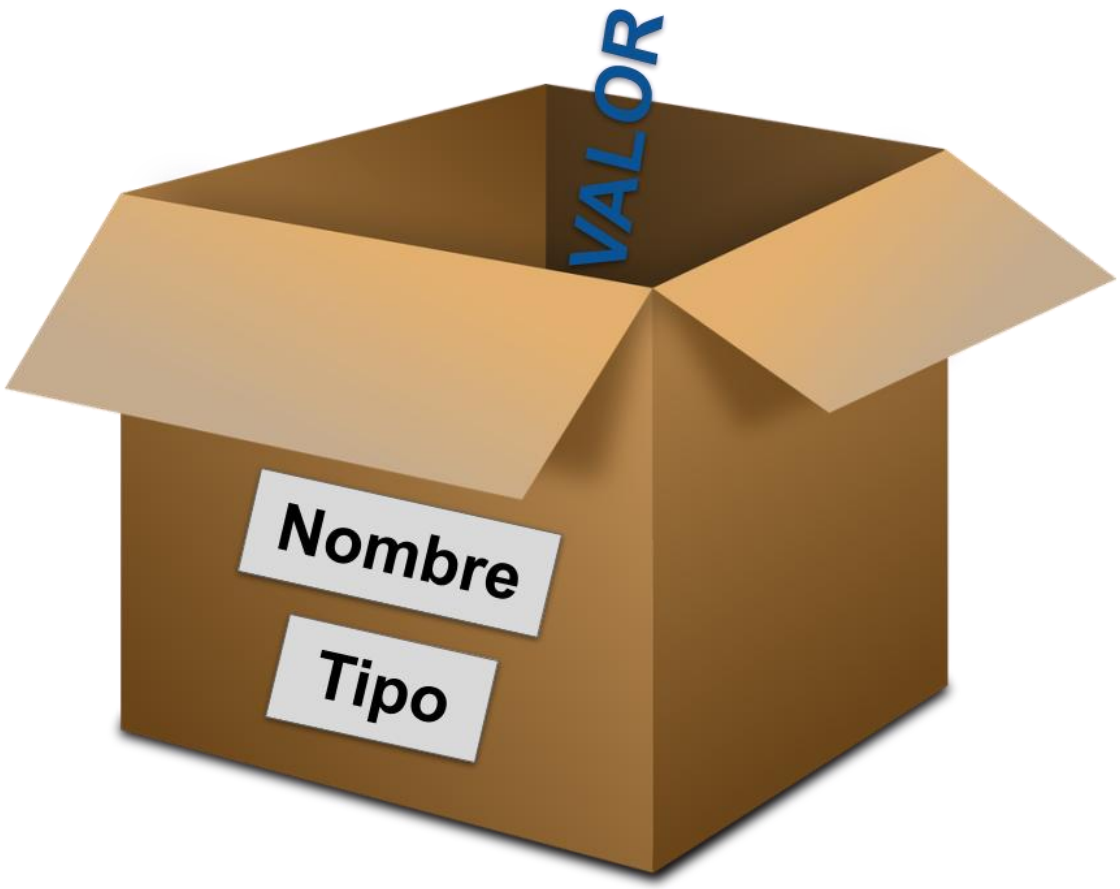
Sintaxis, operadores y tipos de datos

Variables



En este tema vamos a empezar a trabajar con variables, que son la forma que tenemos de almacenar datos en nuestro programa.

Podemos imaginar una variable como una caja a la que le damos un nombre y que es capaz de almacenar un valor, que puede cambiar. Las variables tienen también un tipo, que en Python viene definido por el valor que le damos.

Aunque no tenemos que decirlo explícitamente, en Python las variables tienen tipo.

En la siguiente línea estamos creando un variable y le damos el valor **Pepe**

```
persona = "Pepe"
print("Hola")
print(persona)
```

Llamamos asignación al hecho de dar a una variable un valor dado.

Diremos que hemos declarado la variable **persona** y le asignamos su valor.

Posteriormente, podemos cambiar su valor sin más que asignarle otro nuevo.

Vemos cómo distinguimos cuando queremos imprimir un texto literal, y el valor de la variable.

Para indicar el valor de las variables de tipo texto ponemos comillas simples **'** o dobles **"**.

Los nombres de las variables pueden estar formados por letras (mayúsculas y minúsculas, entre las que se distingue), números y el signo "_" de guion bajo, pero no pueden contener espacios, ni empezar con números. Se recomienda que no se usen letras con acentos o ñ, si bien, todo esto depende más del editor que usemos, y cómo codifique el fichero.

Se recomienda usar nombres claros y descriptivos. Podemos usar guion bajo o separarlas por las letras mayúsculas/minúsculas (a esta técnica se la llama camelCase). Por ejemplo:

```
nombrePersona = "Pepe"
```

ó

```
nombre_persona = "Pepe"
```

De la misma forma, aunque no es obligatorio se suele espaciar las asignaciones y las operaciones para hacerlos más legibles.

Palabras reservadas

Existen ciertas palabras que tienen un uso concreto en Python y por tanto no podemos usarlas como nombres de variables. Puedes encontrar el listado aquí para la versión 3.13 de Python.

Palabras reservadas

False	await	else	import	pass
Nose	break	except	in	raise
True	class	finally	is	return
and	continue	for	lamda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Podemos ver la lista de las palabras reservadas para nuestra versión de Python usando **help('keywords')**.

Entrada

Ya que tenemos dónde guardar datos, vamos a ver cómo podemos pedirlos al usuario usando el teclado para ello.

La función que nos permite hacerlo se llama **input** (poca imaginación con el nombre...), y podemos indicar que queremos que se muestre un texto cuando se pide el dato.

```
nombreUsuario = input("¿Cómo te llamas? ")
print("Hola " + nombreUsuario)
```

Recuerda dejar un cierto espacio entre el final de la pregunta y nuestra respuesta. Podemos usar un espacio, o hacer que se salte a la línea siguiente con **\n**

La función **input** siempre nos va dar un valor de tipo **cadena**, independientemente de lo que introduzca el usuario, por lo que deberemos convertirlos al tipo de datos que necesitemos.

```
nombreUsuario = input("¿Cómo te llamas?\n")
print("Hola " + nombreUsuario)
```

Tipos

Cuando asignamos un valor a una variable, le estamos dando un tipo a la variable. Los tipos pueden ser numéricos, que podrán ser **enteros** o **decimales**, de tipo **cadena**, o **booleanos** (de valor lógico) que sólo pueden ser ver **True** o **False**.

En Python las variables tienen un tipo que se infiere a partir del contexto en la asignación, es decir, no es necesario explícitamente decírselo.

Para expresar **números decimales** usaremos el **.** como separador decimal:

```
# variable decimal o float
pi = 3.1416
```

También podemos usar **variables lógicas o booleanos** que podrán ser **True** o **False** (hay que escribirlas así):

```
# variable booleana o lógica
respuesta = True
```

Cuando asignamos un **Texto** usamos las **comillas**, pero no para los números:

```
edad = 10
nombre = "Juan"
print(nombre)
print(" tiene ")
print(edad)
```

Dado que muchas veces necesitamos imprimir varias cosas de manera consecutiva, podemos separar los valores por comas. El ejemplo anterior quedaría así:

```
edad = 10
nombre = "Juan"
print(nombre, " tiene ", edad)
```

Conviene destacar que es diferente hacer:

```
a = 10
```

que:

```
a = "10"
```

Con el primero podremos hacer **operaciones aritméticas**:

```
a = 10
b = 15
c = a + b
print(c)
```

Obtenemos el esperado resultado de **25**.

Pero si usamos **cadenas**, el resultado puede no ser el esperado:

```
a = "10"
b = "15"
c = a + b
print(c)
```

Obtenemos **1015** que es el resultado de concatenar las dos cadenas

También podemos hacer todas estas operaciones en la consola de Python:

```
>>> a = 10
```

Si necesitamos crear una cadena que incluya finales de línea ('\n') podemos delimitarlas con **'''** (como los comentarios extensos):

```
>>> texto = '''En un lugar de la Mancha
De cuyo nombre no quiero acordarme...'''
>>> print(texto)
```

Dentro de una cadena podemos incluir algunos caracteres especiales como son:

Carácter	Descripción
'\n'	Final de línea
'\t'	Tabulador
'\r'	Vuelve a principio de línea, pero sin pasar a la siguiente línea
'\\'	El carácter \
'\''	La comilla simple
'\"'	La comilla doble

Otra forma de hacerlo más sencillo, en el caso de las comillas simples y dobles es aprovechando que podemos delimitar las cadenas con un tipo u otro de comillas. Por ejemplo:

```
>>> print(" ' ") # imprime una comilla simple '
 '
>>> print(' " ') # imprime una comilla simple "
 "
```

Si en algún caso tuviéramos que incluir estos caracteres, pero no queremos que tenga ese sentido podemos evitarlo añadiendo una letra **'r'** antes de la primera comilla, lo que evita que se escapen caracteres:

```
print(r'a\na') # se imprime en 1 sola línea
```

Como hemos visto, también existen las variables de tipo booleano, que sólo pueden contener 2 valores **True** o **False**, que han de ser escritas de esta manera:

```
isOpen = True

isVisible = False
```

Asignaciones múltiples

Podemos asignar el valor a varias variables en una misma línea, sin más que separar los nombres a un lado del igual por comas, y también los valores al otro lado:

```
edad, nombre = 10, "Juan"

valor1 = valor2 =valor3 = 6
```

Comprobación de tipos

Podemos comprobar el tipo de cualquier variable usando la función **type()**.

Así:

```
>>> text = 'hola'
>>> precio = 3.14
>>> posicion = 100
>>> print(type(precio))
<class 'float'>
>>> print(type(text))
<class 'str'>
>>> print(type(posicion))
<class 'int'>
```

Podemos comprobar si es de un tipo u otro usando la función **isinstance(variable, tipo)**, por ejemplo así:

```
>>> print(isinstance(posicion,int))
True
>>> print(isinstance(posicion,float))
False
>>> print(isinstance(precio,float))
True
>>> print(isinstance(text,str))
True
```

Indicación de Tipos: Annotations

A partir de la versión 3.0 de Python, podemos indicar el tipo que va a tener una variable. Sigue sin ser necesario hacerlo, pero aporta ventajas como el evitar errores en nuestro código.

```
>>> text: str = 'hola'
>>> precio: float = 3.14
>>> posicion: int = 100
>>> condición: bool = True
```

Tipos de variables

A veces, es necesario convertir el valor de una variable a otro tipo. Veamos un ejemplo

```
nombreUsuario = input("¿Cómo te llamas?")
print("Hola ",nombreUsuario)
fechaNacimiento = input("¿En que año naciste?")
edad = 2020 - fechaNacimiento
print("Tu edad es ",edad)
```

Al ejecutarlo, encontramos un error que viene de que la función **input** siempre guarda el valor como una cadena y Python no sabe cómo usar un número con una cadena. Por ello, lo que tenemos que hacer, es guardar el valor del año de nacimiento como un entero. Para ello usaremos la función **int()** que permite convertir el valor de una variable a un tipo entero:

```
nombreUsuario = input("¿Cómo te llamas? ")
print("Hola ",nombreUsuario)
fechaNacimiento = int(input("¿En que año naciste? "))
edad = 2020 - fechaNacimiento
print("Tu edad es ",edad)
```

Existen otras funciones de conversión entre tipos:

Conversión	Tipo final
int()	a entero
float()	a decimal
str()	a cadena
bool()	a booleano

Comprobación de tipos numéricos

Podemos comprobar si el valor de una variable numérica es entero o no. Para ello, usaremos la función **is_integer()** de la siguiente forma:

```
x = 1.23
x.is_integer()


y= 1.0
y.is_integer()
```

Excepciones

A veces, nos podemos encontrar con errores de ejecución en nuestro código y debemos de estar prevenidos. Pueden ser debidos a errores del código y como programadores,

los iremos corrigiendo, mejorando así la calidad del código.

También hay errores que se pueden producir por circunstancias externas, como por ejemplo cuando no podemos acceder a un archivo, o cuando hay un problema de acceso a la red que no está disponible. Son lo que se conoce como **errores en tiempo de ejecución** (**Runtime errors** en inglés) y un buen programa siempre tiene que estar protegido contra ellos.

También es necesario proteger nuestros programas contra el mal uso de los usuarios, que pueden ser mal intencionados o por despiste.

No hay que confundir estos errores con los errores de sintaxis por código mal formado. Existe un mecanismo para tratar estos errores en tiempo de ejecución, cuando nuestro programa se está ejecutando.

Cuando se produce un error de estos, decimos que se genera una **excepción**, y cuando se produce una, nuestro programa se detiene en esa línea y se para.

Podemos hacer que nuestro código capture estas excepciones y así evitar que nuestro programa finalice. Para ello usaremos la estructura **try: ... except :**, que se divide en 2 partes, a las que llamaremos bloques:

- Ponemos **try:** antes de la parte inicial donde pensamos que se puede producir la excepción.
- Ponemos **except:** para indicar el código que queremos que se ejecute si se produce la excepción.

Para delimitar claramente el código de cada parte, añadimos unos espacios (normalmente 4) antes de las líneas de código de cada bloque.

Python no utiliza delimitadores para indicar los bloques de código como otros lenguajes (**{...}** en C, C++ o Java), si no que la **indentación de las líneas** (los espacios iniciales), marca estos bloques.

```
# Comprobar si una cadena es o no un número
s = input('Introduzca un número  entero: ')
try: # Comienzo del bloque try
    valorInt = int(s)  # Se produce una excepción
    print('Es entero') # Esta línea no se ejecuta
except: # Comienzo del bloque except
    print('No es entero')

print('Hemos terminado') # Esta línea no está dentro de los bloques
```

Existen diferentes tipos de excepciones, según el error que se produzca. Las que se generan por conversión, decimos que son de tipo **ValueError**

Vamos a depurar este ejemplo para ver cómo se produce el error paso a paso. Para ello,

usaremos también la opción de **Entrando** del menú de depuración:

O podemos hacer que entre dentro de la ejecución de una línea que puede ser, una función o puede ser un código, un poquito más complejo usando la opción **entrando**.

```
nombre = input('¿Cómo te llamas? ')
print(f'Hola {nombre}')
strEdad = input('¿Cuantos años tienes? ')

try:
    edad = int(strEdad)
    cadena = f'Tienes {edad} años y naciste en el año {2021 - edad}'
    print(cadena)
except:
    print('Se ha producido un error en la conversión')

print('Adios')
```

También podemos lanzar una excepción si se ha producido un error ¿Por qué lanzar una excepción en lugar de hacer algo para corregirlo? porque quizás, no está en nuestra mano el evitar el error.

Imagina que hemos hecho un código que calcula el precio de la cuota de una hipoteca y que el usuario introduce un importe con valor negativo, para el que claramente no tiene sentido. Si no podemos volver a pedir que nos proporcionen un valor correcto, al menos podemos lanzar una excepción indicando el error en los valores proporcionados.

Para hacerlo, usaremos la palabra reservada **raise** seguida de **Exception** a la que le podemos pasar un texto explicativo. Por ejemplo, si queremos hacerlo en el caso anterior del cálculo para un capital negativo haríamos:

```
## Comprobamos que el valor no es válido
raise Excepcion('El capital debe ser mayor que 0')
```

Cuando lanzamos una excepción, en alguna parte del programa se tendrá que capturar (con un try/except) o de lo contrario el programa se detendrá mostrando el error/Excepción.

Operadores Aritméticos

Entre los valores numéricos podemos hacer operaciones matemáticas. Estos son los operadores aritméticos ya definidos:

Operador	Operación
+	suma
-	resta
/	división (con decimales en el resultado)
//	división entera
%	módulo o resto de la división
**	potencia

Podemos hacer operaciones entre números y entre variables usando estos operadores.

Algo muy frecuente es que queramos incrementar o decrementar el valor de una variable, para ello podemos hacer una versión simplificada, poniendo detrás del operador el signo "=". Así: "a = a + 10" se puede simplificar a "a += 10", pudiendo hacerse con todos los operadores:

```
a = 10
a += 5
a *= 10
a //= 7
```

Python no tiene límite en cuanto al tamaño máximo que puede usar para números enteros:

```
a = 10 ** 1000
print(a)
```

Cuando operamos con variables de distintos tipos pero que sean numéricos (bool, int y float), se producen conversiones pasando al tipo más complejo, según la siguiente tabla:

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

Ahora ya podemos empezar a hacer una calculadora que sume 2 números, pero ya vimos que tenemos que darnos cuenta de que las cadenas se suman de forma diferente…

```
a = "10"
b = "15"
c = a + b
print(c)
```

Decimos que se concatenan. Para sumarlos correctamente, los convertiremos en enteros.

```
# Programa que realiza la suma de dos valores
a = input('numero 1 ')  # pedimos el primer número
b = input('numero 2 ')  # pedimos el segundo número
suma = int(a) + int (b)  # calculamos la sumas
print (suma)  # imprimimos su valor
```

Ya que hemos visto que hay algo parecido a "sumar" cadenas, vamos a ver que ¡también podemos multiplicarlas! ...

```
cadena = 'hola'
queSera = cadena*10
print(queSera)
```

Funciones aritméticas

Veamos algunas funciones aritméticas:

Función	Definición
round	Redondeo: de un valor decimal al menor entero
abs	Valor absoluto
divmod(dividendo,diviso)	Devuelve 2 valores: cociente entero y resto
pow(base,exponente)	base elevado a exponente

Un ejemplo de `divmod` nos permite ver que en Python, una función puede devolver varios valores:

```
cociente, resto = divmod(10,7)
```

Excepciones

En los cálculos matemáticos también se pueden producir **excepciones** que debemos tratar de la misma forma que en las **conversiones**.

Por ejemplo, al dividir por 0 se produce una excepción de tipo **ZeroDivisionError**, que si no capturamos con un **try/except**, detendrá nuestro programa:

```
a = 0
b = 0
c = a / b
```

String (cadenas)

Existen multitud de operaciones que podemos hacer sobre una cadena. Veamos algunas:

len - Longitud

Podemos ver la longitud de una cadena con la función **len()**:

```
palabra = input('Introduzca una palabra')
print('la palabra: ' + palabra + ' mide ')
print(len(palabra))
```

replace - Reemplazar

Sustituye todas las repeticiones de una cadena en otra dada, sin cambiar la original:

```
cadena = 'parapapa'
nueva_cadena = cadena.replace('a','e')
print(cadena + ' -> ' + nueva_cadena)
```

strip - limpiar

Elimina los caracteres "Espacio, tabuladores y '\n' del principio y final de una cadena.:

```
cadena = ' hola esto es una prueba '
nueva_cadena = cadena.strip()
print(cadena + ' -> ' + nueva_cadena)
```

```
cadena = ' hola esto es una prueba\nDe varias líneas\n'
nueva_cadena = cadena.strip('\n')
print(cadena + ' -> ' + nueva_cadena)
```

También podemos hacer que se limpie sólo la parte izquierda con **lstrip** o la derecha con **rstrip**

lower y upper- Minúsculas y mayúsculas

Las funciones **lower** y **upper** convierten a minúsculas y mayúsculas todos los caracteres.

También existen **capitulize**, **title** y **swapcase** que se deja como ejercicio averiguar su utilidad...

Formato

Hay veces que nos interesa generar una salida por **print** en la que se mezclan diferentes tipos de variables

A medida que ha ido evolucionando Python, han ido apareciendo diferentes formas, pero a día de hoy, la más utilizada es la que se conoce como **f-string** (desde Python 3.6). Indicamos que vamos a usarla anteponiendo una **f** a la comilla inicial, y definimos el hueco en el que se pondrá la variable, insertando la variable rodeada de unas llaves **{variable}**. No es necesario que hagamos conversión de la variable a cadena:

```
nombre = 'Pepe'
edad = 20

print(f'{nombre} tiene {edad} años')
```

Dentro de las **f-string** se pueden incluir **expresiones**:

```
print(f'{nombre} tiene {edad} años, naciste en {2025-edad}')
```

También podemos **alinear decimales y números** de la siguiente manera:

- Usaremos ":" después de la variable indicando el formato
- número de cifras totales (enteras + decimales) y tras un punto "." el número de decimales.
- Terminaremos la expresión con una "f"

```
valor1 = 1.45
valor2 = 45.45
print(f'{valor1:6.3f}')
print(f'{valor2:6.3f}')
```

```
 1.450
45.450
```

Por si los encuentras en el código, otros formatos más antiguos pero que dan el mismo resultado son:

```
print("{} tiene {} años".format(nombre,edad))
print("%s tiene %d años"%(nombre,edad)) # %s indica que nombre es cadena y %d
que edad es entero
```

Unicode

Podemos acceder a cualquier carácter Unicode, si conocemos su código usando la función **chr(código)**

Por ejemplo:

```
>>>chr(0x1F415)
' 🐕 '
>>> chr(0x1F30E)
' 🌎 '
>>> chr(0x1F5A5)
' 🖥 '
>>> chr(0xFE0F)
''
>>> chr(0x1F4BD)
' 💽 '
```