Python

Tipos y operaciones:

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

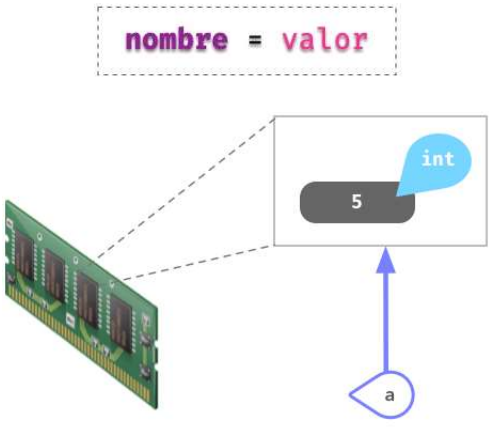
Comentarios:

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

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

Variables:

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



Reglas para nombrar variables

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

1. Sólo pueden **contener los siguientes caracteres**:
    - Letras minúsculas.

- Letras mayúsculas.
- Dígitos o número.
- Guiones bajos (_).

2. Deben **empezar con una letra o un guión bajo**, nunca con un número.
3. No pueden ser una **palabra reservada** del lenguaje («keywords»).

Palabras reservadas:

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

Los nombres de variables son «case-sensitive». Por ejemplo, stuff y Stuff son nombres diferentes.

Constantes:

Podríamos decir que es un tipo de variable pero que su valor no cambia a lo largo de nuestro programa. Por ejemplo la velocidad de la luz. Sabemos que su valor es constante de 300.000 km/s. En el caso de las constantes utilizamos mayúsculas. Por ejemplo: LIGHT_SPEED.

Asignación

En Python se usa el símbolo = para asignar un valor a una variable:

Asignación de valor a nombre de variable

Algunos ejemplos de asignaciones a variables:

```
total_population = 157_503
avg_temperature = 16.8
city_name = 'San Cristóbal de La Laguna'
```

Algunos ejemplos de asignaciones a constantes:

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

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

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

Asignando una variable a otra variable

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

```
people = 157503
total_population = people
print(total_population)
```

Conocer el tipo de una variable

Podemos usar la función type() para conocer el tipo de una variable.

Tipos de datos

Booleanos

Las variables booleanas sólo pueden tomar dos valores discretos: verdadero o falso. `True` o `False`

Operaciones

Podemos usar los tres operadores lógicos más usados:

- and
- or
- not

OR

p	q	p or q
0	0	0
0	1	1
1	0	1
1	1	1

AND

p	q	p and q
0	0	0
0	1	0
1	0	0
1	1	1

NOT

p	not p
0	1
1	0

Operadores de comparación

Para usar las expresiones de comparación es fundamental conocer los operadores que nos ofrece Python:

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

Enteros

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

Dos detalles a tener en cuenta:

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

Operaciones con enteros

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

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

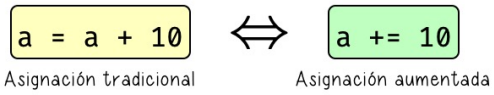
Es de buen estilo de programación dejar un espacio entre cada operador. Además hay que tener en cuenta que podemos obtener errores dependiendo de la operación (más bien de los operandos) que estemos utilizando, como es el caso de la división por cero.

Igualmente es importante tener en cuenta la prioridad de los distintos operadores:

Prioridad	Operador
1 (mayor)	`()`
2	`**`
3	`-a +a`
4	`* / // %`
5 (menor)	`+ -`

Asignación aumentada

Python nos ofrece la posibilidad de escribir una asignación aumentada mezclando la asignación y un operador.



De este mismo modo, podemos aplicar un formato compacto al resto de operaciones

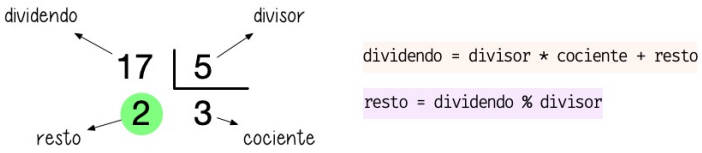
Supongamos que disponemos de 100 vehículos en stock y que durante el pasado mes se han vendido 20 de ellos. Veamos cómo sería el código con asignación tradicional vs. asignación aumentada:

```
total_cars = 100
sold_cars = 20
total_cars = total_cars - sold_cars
print(total_cars)
```

```
total_cars = 100
sold_cars = 20
total_cars -= sold_cars
print(total_cars)
```

Módulo

La operación módulo (también llamado resto), cuyo símbolo en Python es %, se define como el resto de dividir dos números. Veamos un ejemplo para entender bien su funcionamiento




```
dividendo = 17
divisor = 5

cociente = dividendo // divisor  # división entera
resto = dividendo % divisor
print(cociente)
print(resto)
```

Potencia

Para elevar un número a otro en Python utilizamos el operador de exponenciación **:

```
4 ** 3

4 * 4 * 4
```

Se debe tener en cuenta que también podemos elevar un número entero a un número decimal. En este caso es como si estuviéramos haciendo una raíz 2. Por ejemplo:

$$4^{\frac{1}{2}} = 4^{0.5} = \sqrt{4} = 2$$

```
4 ** 0.5
```

Valor absoluto

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

```
abs(-1)

abs(1)

abs(-3.14)

abs(3.14)
```

Decimales - Flotantes

Los números en coma flotante tienen parte decimal. Veamos algunos ejemplos de flotantes en Python.

```
4.0
4.
```

Conversión de tipos

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

Conversión implícita

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

```
print(True + 25)

print(7 * False)

print(True + False)

print(21.8 + True)

print(10 + 11.3)
```

Podemos resumir la conversión implícita en la siguiente tabla:

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

Se puede ver claramente que la conversión numérica de los valores booleanos es:

True → 1

False → 0

Conversión explícita¶

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

bool() Convierte el tipo a booleano.

int() Convierte el tipo a entero.

float() Convierte el tipo a flotante.

str() Convierte el tipo a Cadena o String

Veamos algunos ejemplos de estas funciones:

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

Para obtener el tipo de una variable ya hemos visto la función `type()`

Pero también existe la posibilidad seguimos comprobar el tipo que tiene una variable mediante la función `isinstance()`:

```
>>>
is_raining=False
isinstance(is_raining, bool)
True
sound_level=3
isinstance(sound_level, int)
True
temperature=28.3
isinstance(temperature, float)
True
```

Bases (…)

Los valores numéricos con los que estamos acostumbrados a trabajar están en base 10 (o decimal). Esto indica que disponemos de 10 «símbolos» para representar las cantidades. En este caso del 0 al 9.

Pero también es posible representar números en otras bases. Python nos ofrece una serie de prefijos y funciones para este cometido.

Base binaria

Cuenta con 2 símbolos para representar los valores: 0 y 1.

Prefijo: 0b

```
0b1001
0b1100

Función: bin()

bin(9)
bin(12)
```

Base octal

Cuenta con 8 símbolos para representar los valores: 0, 1, 2, 3, 4, 5, 6 y 7.

Prefijo: 0o

```
0o6243

0o1257

Función: oct()

>>>

oct(3235)

oct(687)
```

Base hexadecimal

Cuenta con 16 símbolos para representar los valores: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F.

Prefijo: 0x

```
>>>

0x7F2A

0x48FF

Función: hex()

>>>

hex(32554)          # '0x7f2a'

hex(18687)          # '0x48ff'
```

Las letras para la representación hexadecimal no atienden a mayúsculas y minúsculas.

Cadenas de texto – Strings

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

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

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

Puede surgir la duda de cómo incluimos comillas simples dentro de la propia cadena de texto. Veamos soluciones para ello:

Comillas simples escapadas

```
cadena = 'Los llamados \'strings\' son secuencias de caracteres'
```

Comillas simples dentro de comillas dobles

```
Cadena = "Los llamados 'strings' son secuencias de caracteres"
```

Comillas triples

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

```
poem = '''To be, or not to be, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles'''
```

Cadena vacía

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

```
cadena = “”
```

Conversión

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

```
str(True)
str(10)
str(21.7)
```

Secuencias de escape

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

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

```
msg = 'Primera línea\nSegunda línea\nTercera línea'

 print(msg)

Primera línea

Segunda línea

Tercera línea


# Tabulador

msg = 'Valor = \t40'

print(msg)

Valor =     40
```

Más sobre `print()`

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

```
msg1 = '¿Sabes por qué estoy aquí?'
msg2 = 'Porque me apasiona'

print(msg1, msg2)
¿Sabes por qué estoy aquí? Porque me apasiona

print(msg1, msg2, sep='|')
¿Sabes por qué estoy acá?|Porque me apasiona

print(msg2, end='!!')
Porque me apasiona!!
```

Más información: https://docs.python.org/es/3/library/functions.html#print

Leer datos desde teclado

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

```
name = input('Introduzca su nombre: ')
Introduzca su nombre: Sergio
print(name)
'Sergio'
type(name)
str

age = input('Introduzca su edad: ')
Introduzca su edad: 41
print(age)
'41'
```

```
type(age)
str
```

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

Operaciones con «strings»

Combinar cadenas

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

```
proverb1 = 'Cuando el río suena'
proverb2 = 'agua lleva'

print(proverb1 + proverb2)


print(proverb1 + ', ' + proverb2)  # incluimos una coma
```

Repetir cadenas

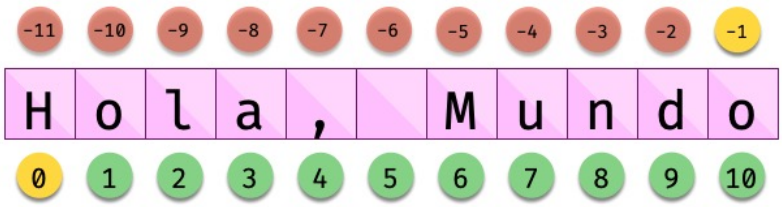
Podemos repetir dos o más cadenas de texto utilizando el operador *:

```
reaction = 'Wow'

print(reaction * 4)
```

Obtener un carácter

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



Veamos algunos ejemplos de acceso a caracteres:

```
sentence = 'Hola, Mundo'

print(sentence[0])
print(sentence[-1])
print(sentence[4])
print(sentence[-5])
print(sentence[50])
```

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

```
song = 'Hey Jude'
song[4] = 'D'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Trocear una cadena

Es posible extraer «trozos» («rebanadas») de una cadena de texto 2. Tenemos varias aproximaciones para ello:

`[:]` Extrae la secuencia entera desde el comienzo hasta el final. Es una especia de copia de toda la cadena de texto.

`[start:]` Extrae desde start hasta el final de la cadena.

`[:end]` Extrae desde el comienzo de la cadena hasta end menos 1.

`[start:end]` Extrae desde start hasta end menos 1.

`[start:end:step]` Extrae desde start hasta end menos 1 haciendo saltos de tamaño step.

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

```
proverb = 'Agua pasada no mueve molino'

print(proverb[:])
'Agua pasada no mueve molino'

print(proverb[12:])
'no mueve molino'

print(proverb[:11])
'Agua pasada'

print(proverb[5:11])
'pasada'

print(proverb[5:11:2])
'psd'
```

Longitud de una cadena

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

```
proverb = 'Lo cortés no quita lo valiente'
print(len(proverb))
30
```

Pertenencia de un elemento

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

```
proverb = 'Más vale malo conocido que bueno por conocer'

print('malo' in proverb)
True
print('regular' in proverb)
False
```

Dividir una cadena

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

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

Limpiar cadenas

Cuando leemos datos del usuario o de cualquier fuente externa de información, es bastante probable que se incluyan en esas cadenas de texto, caracteres de relleno 3 al comienzo y al final. Python nos ofrece la posibilidad de eliminar estos caracteres u otros que no nos interesen.

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

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

```
serial_number = '\n\t   \n 48374983274832    \n\n\t   \t   \n'

print(serial_number.strip())
'48374983274832'
```

A continuación vamos a hacer «limpieza» por la izquierda (comienzo) y por la derecha (final) utilizando la función lstrip() y rstrip() respectivamente:

```
print(serial_number.lstrip())
print(serial_number.rstrip())
```

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

```
print(serial_number.strip('\n'))
```

Realizar búsquedas

Aunque hemos visto que la forma pitónica de saber si una subcadena se encuentra dentro de otra es a través del operador in, Python nos ofrece distintas alternativas para realizar búsquedas en cadenas de texto.

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

print(lyrics.startswith('Quizás'))
print(lyrics.endswith('Final'))
```

Encontrar la primera ocurrencia de alguna subcadena:

```
print(lyrics.find('amor'))            #93
print(lyrics.index('amor'))           #93
print(lyrics.find('universo'))        #-1
```

Contabilizar el número de veces que aparece una subcadena:

```
print(lyrics.count('tu'))             #3
```

Reemplazar elementos

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

```
proverb = 'Quien mal anda mal acaba'

print(proverb.replace('mal', 'bien'))
#'Quien bien anda bien acaba'

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

Mayúsculas y minúsculas

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

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

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

proverb.capitalize()
'Quien a buen árbol se arrima buena sombra le cobija'

proverb.title()
```

```
'Quien A Buen Árbol Se Arrima Buena Sombra Le Cobija'

proverb.upper()
'QUIEN A BUEN ÁRBOL SE ARRIMA BUENA SOMBRA LE COBIJA'

proverb.lower()
'quien a buen árbol se arrima buena sombra le cobija'

proverb.swapcase()
'QUIEN A BUEN ÁRBOL SE ARRIMA bUENA sOMBRA LE COBIJA'
```

Identificando caracteres

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

Veamos algunas de estas funciones:

Detectar si todos los caracteres son letras o números

```
'R2D2'.isalnum()
True
'C3-PO'.isalnum()
False
```

Detectar si todos los caracteres son números

```
'314'.isnumeric()
True
'3.14'.isnumeric()
False
```

Detectar si todos los caracteres son letras

```
'abc'.isalpha()
True
'a-b-c'.isalpha()
False
```

Detectar mayúsculas/minúsculas

```
'BIG'.isupper()
True
'small'.islower()
True
'First Heading'.istitle()
True
```

Interpolación de cadenas

En este apartado veremos cómo interpolar valores dentro de cadenas de texto utilizando diferentes formatos. Interpolar (en este contexto) significa sustituir una variable por su valor dentro de una cadena de texto.

Para indicar en Python que una cadena es un «f-string» basta con precederla de una f e incluir las variables o expresiones a interpolar entre llaves {...}.

```
name = 'Elon Musk'
age = 49
fortune = 43_300

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

Caracteres Unicode

Python trabaja por defecto con caracteres Unicode. Eso significa que tenemos acceso a la amplia carta de caracteres que nos ofrece este estándar de codificación.

Supongamos un ejemplo sobre el típico «emoji» de un cohete definido en este cuadro:

La función chr() permite representar un carácter a partir de su código:

```
rocket_code = 0x1F680
rocket = chr(rocket_code)
print(rocket)
```

El modificador \N permite representar un carácter a partir de su nombre:

```
print('\N{ROCKET}')
```

https://unicode.org/emoji/charts/emoji-list.html

Comparar cadenas

Cuando comparamos dos cadenas de texto lo hacemos en términos lexicográficos. Es decir, se van comparando los caracteres de ambas cadenas uno a uno y se va mirando cuál está «antes».

Por ejemplo:

```
'arca' < 'arpa' # 'ar' es igual para ambas
True

'a' < 'antes'
True

'antes' < 'después'
True

'después' < 'ahora'
False

'ahora' < 'a'
False
```

Tener en cuenta que en Python la letras mayúsculas van antes que las minúsculas

Funciones built-in

Funciones «built-in»

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	any()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Documentación: https://docs.python.org/es/3/library/functions.html?highlight=built