
 No description has been provided for this image No description has been provided for this image

Python Basics I

Contenidos

- [Conversión de tipos](#)
- [Input](#)
- [None](#)
- [Sintaxis y best practices](#)

Conversión de tipos

[al índice](#)

Como en otros lenguajes, en Python también tenemos la posibilidad de realizar conversiones entre los tipos de datos. Hasta ahora hemos visto pocos, pero cuando descubramos más tipos de datos y objetos, verás que son muy comunes estas transformaciones.

Un caso muy habitual es leer un archivo datos numéricos, y que Python interprete los números como caracteres. No es un error, pero posteriormente, cuando hagamos operaciones con nuestros números, tendremos errores, ya que en realidad son cadenas de texto. Si forzamos el cambio a numerico, nos evitaremos futuros problemas.

Mucho cuidado en los cambios de tipos. Tenemos que estar seguros de lo que hacemos ya que podemos perder información, o lo más probable, puede dar error, al no ser compatible el cambio.

Veamos como cambiar los tipos

```
In [1]: num_real = 24.69
        print(type(num_real))

        num_entero = int(num_real)
```

```
print(num_entero)
print(type(num_entero))
```

```
<class 'float'>
24
<class 'int'>
```

Perdemos unos decimales, normalmente nada grave, aunque depende de la aplicación.

NOTA: si lo que queremos es redondear, podemos usar la función `round()`

```
In [3]: # Esta funcion tiene dos argumentos: el número y la cantidad de decimal
# Veremos funciones más adelante.
num_real = 24.686
num_redondeado = round(num_real, 2)
```

```
In [4]: print(num_redondeado)
```

```
24.69
```

```
In [5]: num_redondeado = 24.69
num_convertido = round(num_redondeado, 0)
print(num_convertido, type(num_convertido))
```

```
25.0 <class 'float'>
```

```
In [6]: print(int(num_convertido))
```

```
25
```

Para pasar de un **numero a string**, no hay problema

```
In [8]: num_real = 24.69
num_entero = 12

real_str = str(num_real)
entero_str = str(num_entero)

print(real_str, "su tipo ", type(real_str))
print(entero_str, "su tipo ", type(entero_str))
```

```
24.69 su tipo <class 'str'>
12 su tipo <class 'str'>
```

```
In [9]: print(num_real + num_entero)
print(real_str + entero_str)
```

```
36.69
24.6912
```

De **String a un número** tampoco suele haber problema. Hay que tener mucho cuidado con los puntos de los decimales. **Puntos, NO comas**

```
In [12]: mi_cadena = 98
mi_entero = int(mi_cadena)
print(mi_cadena, type(mi_cadena))
print(mi_entero, type(mi_entero))
mi_cadena_real = "98,33"
mi_real = float(mi_cadena_real)
```

```
98 <class 'int'>
```

```
98 <class 'int'>
```

```
-----
---
ValueError                                Traceback (most recent call last)
Cell In[12], line 6
      4 print(mi_entero, type(mi_entero))
      5 mi_cadena_real = "98,33"
----> 6 mi_real = float(mi_cadena_real)

ValueError: could not convert string to float: '98,33'
```

```
In [13]: mi_cadena_real = "98.64"
mi_real = float(mi_cadena_real)
print(mi_real, type(mi_real))
```

```
98.64 <class 'float'>
```

Pasar de **numero a booleano y viceversa**, tambien es bastante sencillo.

Simplemente ten en cuenta que los 0s son `False`, y el resto de numeros equivalen a un `True`

```
In [14]: booleano = bool(123)
print(booleano, type(booleano))
```

```
True <class 'bool'>
```

```
In [15]: booleano = bool(0)
print(booleano, type(booleano))
```

```
False <class 'bool'>
```

En el caso de transformar **String a booleano**, los strings vacíos serán `False`, mientras que los que tengan cualquier cadena, equivaldrán a `True`.

Sin embargo, si la operación es la inversa, el booleano `True` pasará a valer una cadena de texto como `True`, y para `False` lo mismo.

```
In [16]: booleano = bool("")
booleano_2 = bool("Esto es otro tipo de booleano al hacer la conversion")
print(booleano, type(booleano))
print(booleano_2, type(booleano_2))
```

```
False <class 'bool'>
```

```
True <class 'bool'>
```

```
In [17]: bool_str = str(booleano)
bool_2_str = str(booleano_2)
print(bool_str, bool_2_str)
```

False True



No
description
has been
provided for
this image

ERRORES en conversion de tipos

Ojo si intentamos pasar a entero un string con pinta de real. En cuanto lleva el punto, ya tiene que ser un numero real. A no ser que lo pasemos a real y posteriormente a entero (`int()`), o usando el `round()` como vimos anteriormente

```
In [18]: real_str = "68,34"
real = float(real_str)
```

```
-----
---
ValueError                                Traceback (most recent call la
st)
Cell In[18], line 2
      1 real_str = "68,34"
----> 2 real = float(real_str)

ValueError: could not convert string to float: '68,34'
```

Si leemos datos con decimales y tenemos comas en vez de puntos, habrá errores.

```
In [19]: real_str.replace(",", ".")
```

Out[19]: '68.34'

Para solventar esto utilizaremos funciones que sustituyan unos caracteres por otros

```
In [20]: nuevo_str = real_str.replace(",", ".")
print(type(nuevo_str))
```

<class 'str'>

```
In [22]: real = float(nuevo_str)
print(type(real))
```

<class 'float'>

Es fundamental operar con los mismos tipos de datos. Mira lo que ocurre cuando sumamos texto con un número. Da un `TypeError`. Básicamente nos dice que no puedes concatenar un texto con un número entero

In []:

Input

al índice

Esta sentencia se usa en Python para recoger un valor que escriba el usuario del programa. Los programas suelen funcionar de esa manera, reciben un input, realizan operaciones, y acaban sacando un output para el usuario.

Por ejemplo, en la siguiente celda recojo un input, que luego podremos usar en celdas posteriores.

CUIDADO. Si corres una celda con un `input()` el programa se queda esperando a que el usuario meta un valor. Si en el momento en el que está esperando, vuelves a correr la misma celda, se te puede quedar pillado, depende del ordenador/versión de Jupyter. Si eso ocurre, pincha en la celda y dale después al botón de stop de arriba, o sino a Kernel -> Restart Kernel...

```
In [23]: primer_input = input("Escribeme")
```

```
In [24]: print(primer_input)
```

98

```
In [25]: type(primer_input)
```

```
Out[25]: str
```

Puedes poner ints, floats, strings, lo que quieras recoger en texto plano.

Mira que fácil es hacer un chatbot un poco tonto, mediante el que hacemos preguntas al usuario, y almacenamos sus respuestas.

```
In [26]: nombre = input("¿Como te llamas?")
print("Encantado de conocerte", nombre)
preferencias = input("Y, ¿qué te parece este video?")
print("Coincido")
```

Encantado de conocerte Jose
Coincido

None

[al índice](#)

Palabra reservada en Python para designar al valor nulo. `None` no es 0, tampoco es un string vacío, ni `False`, simplemente es un tipo de datos más para representar el conjunto vacío.

```
In [27]: print(None)
```

None

```
In [28]: print(type(None))
```

<class 'NoneType'>

```
In [29]: "" == None
```

Out[29]: False

```
In [30]: 0 == None
```

Out[30]: False

```
In [31]: False == None
```

Out[31]: False

Sintaxis y best practices

[al índice](#)

A la hora de escribir en Python, existen ciertas normas que hay que tener en cuenta:

- Todo lo que abras, lo tienes que cerrar: paréntesis, llaves, corchetes...
- Los decimales se ponen con puntos `.`
- Best practices
 - **Caracteres:** NO se recomienda usar Ñs, acentos o caracteres raros (`ª,º,@,ç...`) en el código. Ponerlo únicamente en los comentarios.
 - **Espacios:** NO usar espacios en los nombres de las variables ni de las funciones. Se recomienda usar guión bajo para simular el espacio. O también juntar las palabras y usar mayúscula para diferenciarlas `miVariable`. Lo normal es todo minúscula y guiones bajos
 - Ahora bien, sí se recomienda usar espacios entre cada comando, para facilitar la lectura, aunque esto ya es más cuestión de gustos.

```
mi_variable = 36 .
```

- Se suelen declarar las variables en minúscula.
- Las constantes (variables que no van a cambiar nunca) en mayúscula.

```
MI_PAIS = "España"
```

- **Cada sentencia en una línea.** Se puede usar el `;` para declarar varias variables, pero no es lo habitual
- **Comentarios:** TODOS los que se pueda. Nunca sabes cuándo otra persona va a coger tu espectacular código, o si tu yo del futuro se acordará de por qué hiciste ese bucle while en vez de un for.
- **Case sensitive:** sensible a mayúsculas y minúsculas. CUIDADO con esto cuando declaremos variables o usemos Strings
- **Sintaxis de línea:** para una correcta lectura del código lo mejor es aplicar sintaxis de línea en la medida de lo posible

```
In [ ]: # A esto nos referimos con sintaxis de línea
lista_compra = ['Manzanas',
                'Galletas',
                'Pollo',
                'Cereales']
```

The Zen of Python

Cualquier consejo que te haya podido dar hasta ahora se queda en nada comparado con los 20 principios que definió *Tim Peters* en 1999. Uno de los mayores colaboradores en la creación de este lenguaje

```
In [32]: import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

In []:

In []: