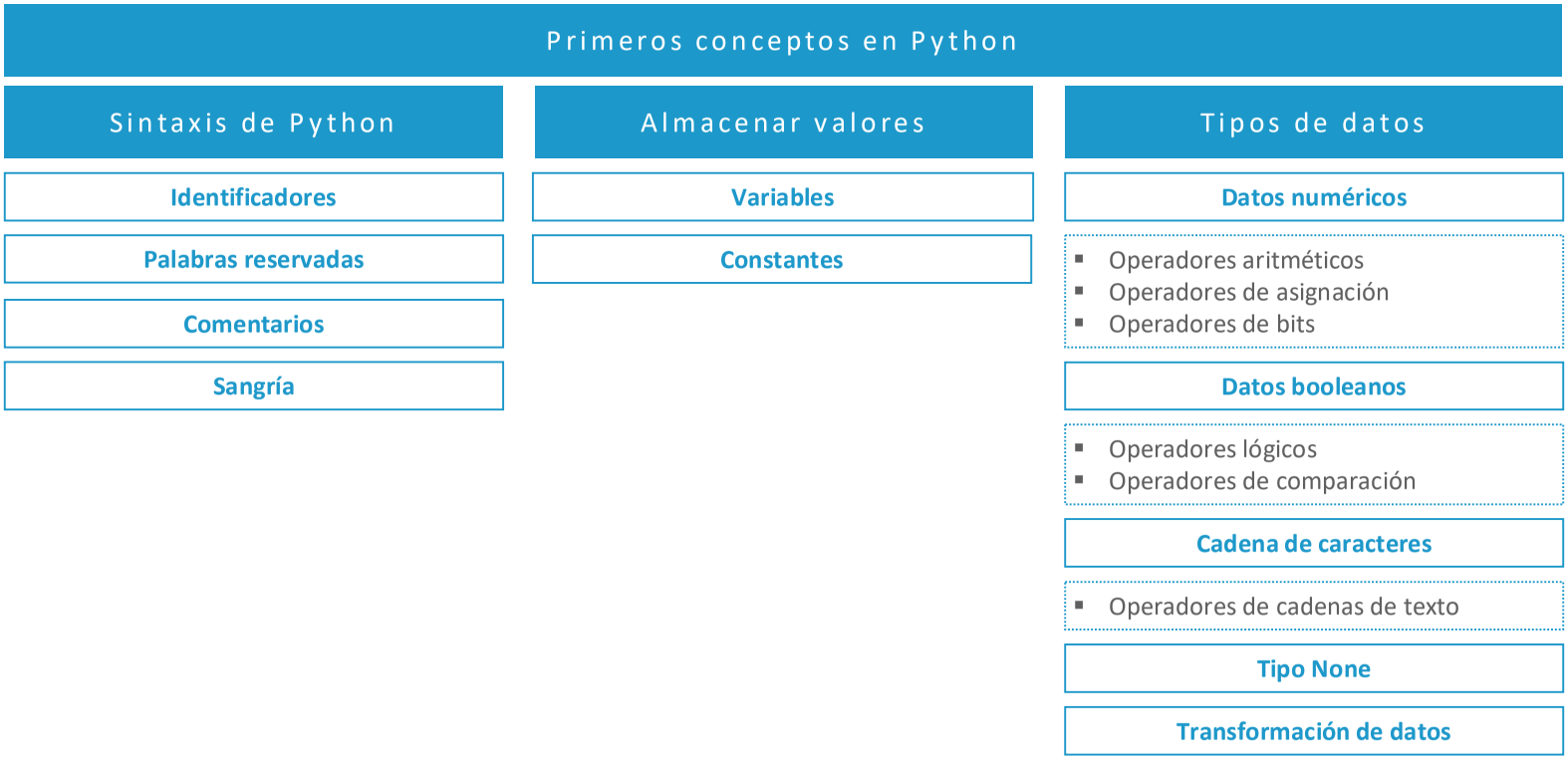
Introducción a la programación y al análisis de datos con Python

Primeros conceptos en Python

Índice

Esquema



Ideas clave

2.1. Introducción y objetivos

En este tema trataremos los primeros conceptos que nos permitirán hacer nuestros programas en Python. En primer lugar, veremos algunas reglas de cómo se deben escribir los diferentes elementos en Python, es decir, su sintaxis. Esto es debido a que Python hace algunos cambios con respecto a otros lenguajes como Java y C para mejorar la legibilidad del código. A continuación, enumeraremos las dos formas que tenemos para almacenar valores en Python y poder usarlos durante la ejecución de nuestros programas. Por último, veremos los tipos de datos básicos que podemos utilizar en Python, algunos de los operadores de cada uno de estos tipos de datos y cómo cambiar el tipo de dato de una variable.

Al finalizar este tema, habrás alcanzado los siguientes objetivos para hacer tus primeros programas en Python:

- Conocer las sintaxis de Python para los identificadores, los comentarios y la sangría.
- Conocer cómo almacenar valores en variables y constantes.
- Comprender los tipos básicos de Python: numéricos, booleanos, cadenas de caracteres y el tipo None.
- Conocer los diferentes operadores que podemos aplicar a cada uno de los tipos de datos.
- Aprender cómo se puede cambiar el tipo de dato de las variables.

2.2. Sintaxis de Python

En este primer apartado, explicaremos algunas normas básicas en la sintaxis de Python para familiarizarnos con el lenguaje. Como explicamos al comienzo del curso, Python sigue una guía de estilo con el objetivo de que los programas sean fáciles de leer. Algunas de las normas que explicaremos aquí han sido creadas con el mismo objetivo. A continuación, detallaremos estas normas básicas.

Identificadores

Los identificadores son nombres que asignaremos a elementos, como funciones, variables, etc., para hacerles referencia más adelante en el código. Estos identificadores tienen que seguir las siguientes reglas:

- Pueden ser una combinación de números (0-9), letras mayúsculas (A-Z), letras minúsculas (a-z) y el símbolo de guion bajo (_).
- No pueden comenzar por un dígito.
- No pueden utilizarse símbolos especiales: @, !, #, etc.
- Pueden tener cualquier longitud.

Hay que decir que Python distingue las mayúsculas de las minúsculas. Esto significa que, si tuviéramos los identificadores `identificador` e `IDENTIFICADOR`, Python los consideraría identificadores diferentes. En el siguiente bloque veremos algunos ejemplos:

```
# Identificadores correctos.
variable = 1
otra_variable = 'Otra'
_mas_variables = [1, 3, 4]
Variable_4 = True

# Identificadores incorrectos.
@variable = 3
2_variable ='Está mal.’
```

Palabras reservadas

Python, como todos los lenguajes de programación, tiene un conjunto de palabras reservadas que no se pueden utilizar como identificadores. Esta lista de palabras reservadas es la siguiente: `and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield`.

Comentarios

Los comentarios son partes del código que el intérprete de Python no ejecuta. Estos comentarios nos permiten escribir aclaraciones en el código para mejorar su interpretabilidad por parte de otros desarrolladores o para nosotros mismos en un futuro. En Python hay dos sintaxis diferentes para escribir los comentarios:

- **Comentarios de línea:** para aquellos comentarios que solo ocupen una línea se utiliza el símbolo # al principio de la misma.
- **Comentarios de bloque:** para los comentarios que ocupen más de una línea se utilizan triples comillas (simples o dobles) al principio y al final del comentario.

A continuación, se muestra un ejemplo de ambos tipos de comentarios:

```
# Ejemplo de comentario de línea.

"""
Ejemplo de comentario de bloque.
Todas las líneas pertenecen al comentario.
"""

'''
Otro ejemplo de
comentario de bloque.
'''
```

Sangría

Una de las diferencias que tiene Python con respecto a otros lenguajes, como C o Java, es que no utiliza llaves ({,}) para diferenciar bloques de código. En Python se utiliza la sangría de bloques, es decir, hacer una separación hacia la derecha del código que pertenece a un bloque. Esta sangría está determinada por 4 espacios según la guía de estilos PEP 8.

Esta forma de definir los bloques de código hace que el código sea más legible. Sin embargo, al ser obligatorio, puede llevar a errores de ejecución si no se ha hecho la sangría correctamente. A continuación, vemos un ejemplo de cómo funciona la sangría dentro de un bloque *if*.

```
# Ejemplo de una sangría:
variable = 1

if (variable == 1):
    print("Aquí pongo el código a ejecutar.")
    print("Tiene una sangría de 4 espacios.")
```

Estas son las normas más importantes de sintaxis que debemos tener en cuenta para comenzar a programar en Python. Durante el curso, incluiremos algunas otras reglas en la sintaxis o en el estilo según vayamos incluyendo nuevos conceptos.

2.3. Almacenar valores

Unas de las principales herramientas que necesitamos en cualquier lenguaje de programación son elementos que nos permitan almacenar valores para usarlos más adelante en nuestro programa. En este apartado explicaremos cómo se deben declarar las variables y las constantes en Python.

Variables

A la hora de implementar nuestros programas será necesario almacenar algunos valores para consultarlos o modificarlos durante la ejecución. Para poder hacerlo utilizaremos las **variables.** Una variable es un identificador al que le asignaremos un valor y, más adelante, llamando a ese identificador podremos consultar el valor.

En Python la creación de una variable se hace a través de una asignación. Haremos una asignación escribiendo el nombre de un identificador, seguido del símbolo = y el valor que deseamos almacenar en la variable:

```
identificador = [valor]
```

A diferencia de otros lenguajes, no es necesario definir el tipo de dato que almacenará la variable. El motivo es que Python infiere este tipo en el momento de ejecutar la asignación y le asignará el tipo de dato que mejor se adapte al valor que hayamos asignado. Por ejemplo, si asignamos a una variable el valor `'Hola mundo!'`, esta variable será de tipo cadena de texto. Además, una misma variable puede almacenar diferentes tipos de datos durante la ejecución.

A continuación, veremos algunos ejemplos de variables y de los tipos de datos que se han asignado. Para preguntar qué tipo de dato ha almacenado una variable, usaremos la función **type:**

```
# Ejemplo de variable:
n = 23
saludo = 'Hola! Qué tal?'
type(n) # Devuelve int
type(saludo) # Devuelve str
```

Constantes

Otro elemento muy utilizado en el desarrollo de programas es el uso de **constantes.** Una constante es un tipo de variable cuyo valor no puede ser modificado durante

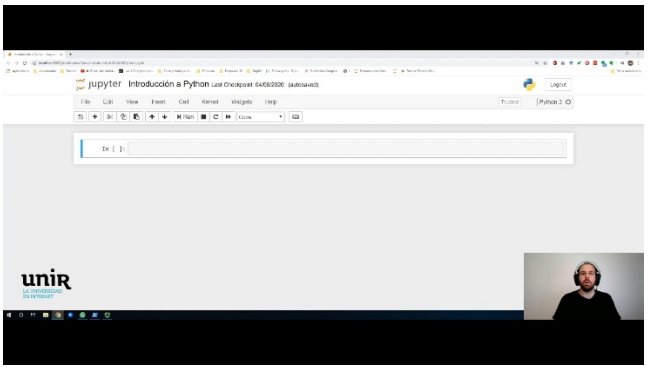
toda la ejecución del programa. Se suele utilizar para almacenar valores que necesitaremos tener disponibles a lo largo de la ejecución.

En Python no existe ninguna palabra reservada que nos permita definir una constante. Sin embargo, existen dos posibles soluciones para declarar constantes. Una de ellas es utilizar una variable, escribiendo el identificador en mayúsculas para que sepamos que esa variable no puede ser modificada. Esta solución se refiere a los estilos y tendremos que estar pendientes de que esa constante nunca cambia de valor.

```
# Ejemplo de constantes:
IVA = 0.21

precio = 25
precio_final = precio + (precio * IVA)

print("El precio final es:", precio_final)
```

Otra solución muy utilizada es crear un *script* de Python, accesible desde nuestro proyecto, donde se almacenen todas las constantes. A continuación, importamos ese *script* con la instrucción **`import`** para poder llamar a esas constantes. El ejemplo anterior, usando un *script* de constantes, sería el siguiente:

```
# Importamos el fichero de constantes.
import Constantes

precio = 25
precio_final = precio + (precio * Constantes.IVA)

print("El precio final es:", precio_final)
```



Vídeo 1. Sintaxis de Python y almacenar valores.

Accede al vídeo a través del aula virtual

2.4. Tipos de datos y operadores

Como hemos visto, cuando declaramos un elemento para almacenar un valor en Python, ya sea una variable o una constante, no necesitamos definir el tipo de dato que almacenará. Sin embargo, es necesario conocer los tipos de datos que podemos encontrarnos en Python para saber qué operadores podemos aplicar. A continuación, describiremos los tipos de datos básicos que encontraremos en este lenguaje y los operadores que podemos aplicar a cada uno de estos tipos.

Datos numéricos

Los primeros tipos de datos que nos encontramos son los tipos numéricos. Dentro de los tipos numéricos encontramos los siguientes tipos más específicos:

- **Enteros** (`int`): `26`, `0b1101` (base binaria), `0x3f4a` (base hexadecimal).
- **Flotante** (`float`): `3.14`, `5.`, `-67.763`
- **Complejos** (`complex`): `0.117j`

Usando estos tipos de datos, podemos aplicar los siguientes tipos de operadores: operadores aritméticos, operadores de asignación y operadores de bits. A continuación, revisaremos estas operaciones y su sintaxis en Python.

Operadores aritméticos

Estos operadores incluyen las operaciones matemáticas básicas:

- **Suma** (+): devuelve como resultado la suma de dos números.

```
3.13 + 6 # Devuelve 9.13
```

- **Resta** (-): devuelve como resultado la resta de dos números.

```
3.13 - 6 # Devuelve -2.87
```

- **Multiplicación** (*): devuelve como resultado la multiplicación de dos números.

```
3.13 * 10 # Devuelve 31.3
```

- **División** (/): devuelve como resultado la división de dos números.

```
3.13 / 10 # Devuelve 0.313
```

- **División entera** (//): devuelve como resultado la división entera de dos números. Es decir, el resultado será la únicamente la parte entera de la división.

```
3 // 10 # Devuelve 0
```

- **Módulo** (%): devuelve como resultado el valor del resto obtenido de la división entera entre dos números.

```
3 % 10 # Devuelve 3
```

- **Exponente** (**): devuelve como resultado el valor exponencial de una base con respecto al exponente:

```
3 ** 2 # Devuelve 9
```

Operadores de asignación

Los operadores de asignación permiten asignar el resultado de la operación a una variable incluyendo el símbolo = en el operador. Estos nos permiten modificar el valor de una variable sin tener que definirla en la parte derecha de la asignación. La lista de operadores de asignación es la siguiente.

- **Asignación simple** (=): asigna a la variable del lado izquierdo el valor definido en la parte derecha.

```
resultado = 10 # resultado vale 10
```

- **Suma y asignación** (+=): el operador suma, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado += 10 # resultado vale 20
```

- **Resta y asignación** (-=): el operador resta, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado -= 10 # resultado vale 0
```

- **Multiplicación y asignación** (*=): el operador multiplica, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado *= 10 # resultado vale 100
```

- **División y asignación** (/=): el operador divide, al valor de la variable, el valor definido en el lado derecho:

```
resultado = 10 # resultado vale 10
resultado /= 10 # resultado vale 1
```

- **División entera y asignación** (//=): el operador realiza la división entera al valor de la variable con respecto al valor definido en el lado derecho:

```
resultado = 14 # resultado vale 14
resultado //= 10 # resultado vale 1
```

- **Módulo y asignación** (%=): el operador asigna a la variable el resto de la división entera entre el valor de la variable y el valor definido en el lado derecho de la operación:

```
resultado = 14 # resultado vale 14
resultado %= 10 # resultado vale 4
```

- **Exponente y asignación** (**=): el operador asigna a la variable el resultado del exponente entre el valor de la variable y el valor de la derecha de la operación:

```
resultado = 3 # El resultado vale 3
resultado **= 2 # El resultado vale 9
```

Operaciones de bits

Otra de las operaciones básicas que podemos realizar en Python con valores numéricos son las operaciones de bits. Estas operaciones solo se pueden aplicar a valores enteros. A continuación, repasaremos estas operaciones:

- **AND** (&): operador lógico *and* a nivel de bits.

```
4 & 5 # El resultado será 4
```

- **OR** (|): operador lógico *or* a nivel de bits.

```
4 | 5 # El resultado será 5
```

- **XOR** (^): operador lógico *xor* a nivel de bits.

```
4 ^ 5 # El resultado será 1
```

- **Mover bits a la izquierda** (<<): operador que mueve todos los bits a la izquierda tantas posiciones como se indique en el lado derecho del operador.

```
4 << 1 # El resultado será 8
```

- **Mover bits a la derecha** (>>): operador que mueve todos los bits a la derecha tantas posiciones como se indique en el lado derecho del operador.

```
4 >> 1 # El resultado será 2
```

Datos booleanos

El tipo de datos booleano es un tipo binario cuyos valores solo pueden ser True o False. Este tipo de datos es muy utilizado en expresiones de control para diferentes sentencias como son el *if* o el *while*. A este tipo de datos se les pueden aplicar los operadores lógicos:

- **AND** (and): operador lógico *and*.

```
True and False # Devolverá False
```

- **OR** (or): operador lógico *or*.

```
True or False # Devolverá True
```

- **NOT** (not): operador de negación lógica.

```
not True # Devolverá False
```

Además de los operadores lógicos, podemos utilizar los operadores de comparación y de identidad:

- **Menor** (<): operador que devuelve `True` si el valor de la izquierda es menor que el valor de la derecha. En caso contrario devolverá `False`.

```
5 < 7 # Devolverá True
```

- **Menor o igual** (<=): operador que devuelve `True` si el valor de la izquierda es menor o igual que el valor de la derecha. En caso contrario devolverá `False`.

```
7 <= 7 # Devolverá True
```

- **Mayor** (>): operador que devuelve `True` si el valor de la izquierda es mayor que el valor de la derecha. En caso contrario devolverá `False`.

```
5 > 7 # Devolverá False
```

- **Mayor o igual** (>=): operador que devuelve `True` si el valor de la izquierda es mayor o igual que el valor de la derecha. En caso contrario devolverá `False`.

```
7 >= 7 # Devolverá True
```

- **Igual** (==): operador que devuelve `True` si el valor de la izquierda es igual que el valor de la derecha. En caso contrario devolverá `False`.

```
5 == 7 # Devolverá False
```

- **Distinto** (!=): operador que devuelve `True` si el valor de la izquierda es distinto que el valor de la derecha. En caso contrario devolverá `False`.

```
7 != 7 # Devolverá False
```

Cadenas de caracteres

Las cadenas de texto son secuencias de caracteres encapsuladas con comillas simples ('') o comillas dobles (""). A las cadenas de texto almacenadas en variables podemos aplicarle diferentes operadores. Para este apartado crearemos una variable llamada *mensaje* e iremos viendo las diferentes operaciones que podemos aplicar:

```
mensaje = "Esto es un mensaje de prueba para el curso de Python."
```

Las cadenas de texto funcionan como una lista de caracteres. Por lo tanto, podemos obtener el carácter que existen en una posición concreta de la cadena. Para ello, aplicamos el operador **[]** indicando la posición a la que queremos acceder dentro de él.

```
mensaje[34] # Devuelve 'e'
```

Este operador no solo nos permite acceder a una única posición, sino que también podemos introducir un rango de caracteres. Para ello usamos este operador separando la posición inicial y la posición final a las que queremos acceder de la siguiente forma:

```
CADENA[POSICION_INICIAL:POSICION_FINAL]
```

Hay que tener en cuenta que esto nos devolverá todos los caracteres desde la `POSICION_INICIAL` hasta la posición anterior a la `POSICION_FINAL`.

```
mensaje[4:15] # Devuelve ' es un mens'
```

Si dejamos la `POSICION_INICIAL` vacía, Python nos devolverá los caracteres desde el comienzo de la cadena hasta la `POSICION_FINAL`. Si, por otro lado, dejamos vacía la `POSICION_FINAL`, nos devolverá los caracteres desde la `POSICION_INICIAL` hasta el final de la cadena de caracteres.

```
mensaje[:20] # Nos devolverá 'Esto es un mensaje d'
mensaje[40:] # Nos devolverá 'so de Python.'
```

En ambas posiciones se pueden incluir números negativos. En estos casos, Python empezará a contar las posiciones desde el lado derecho.

```
mensaje[-10:] # Desde la posición -10 (es decir, 10 desde la derecha)
hasta el final. Nos devolverá 'de Python.'
mensaje[:-10] # Desde la posición inicial, hasta la posición -10 (es
decir, 10 desde la derecha). Nos devolverá 'Esto es un mensaje de prueba
para el curso '
```

Operadores de cadenas de texto

Las cadenas de texto cuentan con los operadores que hemos visto en los otros tipos de datos, aunque el resultado está aplicado al dominio de las cadenas de texto. Estos operadores son:

- **Concatenar** (+): este operador devuelve una cadena de caracteres uniendo los caracteres de dos cadenas.

```
"Hola, " + "estoy bien" # Devolverá 'Hola, estoy bien'
```

- **Multiplicar** (*): este operador nos permite repetir una cadena de caracteres tantas veces como indiquemos en la parte derecha del operador.

```
"Hoy hace sol! " * 3 # Devolverá 'Hoy hace sol! Hoy hace sol! Hoy hace sol! '
```

Además de poder acceder a posiciones concretas de una cadena de texto y los operadores que hemos visto, Python nos proporciona diferentes funciones para obtener información y manipular cadenas de caracteres. A continuación, veremos las más importantes:

- **`len()`**: esta función nos permite obtener la longitud de la cadena de caracteres, es decir, nos devuelve cuantos caracteres están contenidos en la cadena.

```
len(mensaje) # Devuelve 53
```

- **`find()`**: permite obtener la primera posición donde se encuentra la subcadena que pasamos por parámetro dentro de la cadena de texto original. En caso de que la subcadena no exista dentro de la cadena original, nos devolverá -1.

```
mensaje.find('Python') # Devolverá 46
mensaje.find('Java') # Devolverá -1
```

- **`upper()`**: convierte todos los caracteres de la cadena de texto en mayúsculas.

```
mensaje.upper() # Devolverá 'ESTO ES UN MENSAJE DE PRUEBA PARA EL CURSO DE PYTHON.'
```

- **`lower()`**: este operador convierte todos los caracteres de la cadena de texto en minúsculas.

```
mensaje.lower() # Devolverá 'esto es un mensaje de prueba para el curso de python.'
```

- **`replace()`**: nos permite modificar el contenido de la cadena de caracteres. Para ello, introducimos dos parámetros. El primero de ellos contiene la subcadena que queremos sustituir. El segundo contiene la cadena de texto que sustituirá a la primera subcadena. Si la primera subcadena no existe en la cadena de texto original, no habrá ningún cambio.

```
mensaje.replace("curso", "seminario") # Devolverá 'Esto es un mensaje de prueba para el seminario de Python.'
mensaje.replace("Java", "C++") # Devolverá 'Esto es un mensaje de prueba para el curso de Python.'
```

Tipo None

El tipo `None` se utiliza para definir que el valor de una variable es nada o ninguna cosa. Hay que tener cuidado con este tipo de datos, ya que es común utilizarlo cuando queremos declarar una variable, pero no le queremos asignar ningún valor:

```
variable = None
type(variable) # Devolverá que es de tipo NoneType
```

Es importante saber que `None` es un tipo de dato propio con su significado y que lo debemos diferenciar de valores por defecto de otros tipos como el booleano (`False`) o numérico (`0`). En estos casos `None` es un tipo y valor diferente.

```
None == False # Devolverá False
```

Transformación de tipos de datos

Por último, una utilidad muy importante en todos los lenguajes de programación es hacer transformaciones entre tipos de datos. A continuación, vamos a hacer un repaso de las transformaciones de tipos de datos permitidas en Python.

- **Convertir a cadena de caracteres** (`str`): convierte un objeto que se pasa por parámetro a cadena de caracteres.

```
valor = 10
str(valor) # Devolverá '10'
```

- **Convertir a valor entero** (`int`): convierte un objeto que se pasa por parámetro al tipo entero. En caso de ser un número decimal, nos devolverá únicamente la parte entera. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 10.78
valor2 = 'Hola'
int(valor1) # Devolverá 10
int(valor2) # Devolverá un error de tipo ValueError
```

- **Convertir a valor flotante** (`float`): convierte un objeto que se pasa por parámetro al tipo flotante. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

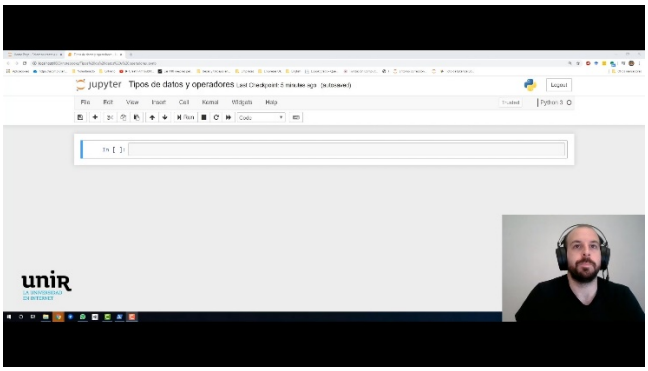
```
valor1 = 30
valor2 = 'Hola'
float(valor1) # Devolverá 30.0
float(valor2) # Devolverá un error de tipo ValueError
```

- **Convertir a valor complejo** (`complex`): convierte un objeto que se pasa por parámetro al tipo complejo. Si el objeto que se pasa no se puede convertir a entero, devolverá un error.

```
valor1 = 10.78
valor2 = 'Hola'
complex(valor1) # Devolverá 10.78 + 0j
complex(valor2) # Devolverá un error de tipo ValueError
```

- **Convertir a valor booleano** (`bool`): convierte un objeto que se pasa por parámetro al tipo booleano. Su funcionamiento es el siguiente:
  - Si no se pasa ningún parámetro, devolverá `False`.
  - En el resto de los casos devolverá `True` excepto si: el valor del parámetro es `0`, si es una secuencia vacía de alguna estructura de datos, si se pasa el tipo `None`, si se pasa el valor `False`.

```
valor1 = 10.78
valor2 = False
valor3 = 0
bool(valor1) # Devolverá True
bool(valor2) # Devolverá False
bool(valor3) # Devolverá False
```



Vídeo 2. Tipos de datos y operadores.

Accede al vídeo a través del aula virtual