

TEMA 1

Introducción a Python

Origen

Python es un lenguaje de programación de alto nivel que ha crecido enormemente en popularidad desde su creación debido a su **simplicidad**, **flexibilidad** y una **comunidad de desarrolladores** altamente activa.

Aunque su curva de aprendizaje es suave al inicio, dominarlo al 100% requiere esfuerzo. Gracias a su gran colección de librerías podemos hacer cosas complejas con pocas líneas de código.

Python fue creado en 1989 por **Guido van Rossum**, un programador holandés. El objetivo principal de Van Rossum era desarrollar un lenguaje que fuera fácil de leer y escribir, reduciendo la complejidad que enfrentaban los programadores con otros lenguajes de la época. En 1991, lanzó la primera versión pública de Python, la versión 0.9.0, que ya incluía características fundamentales como manejo de excepciones y funciones.

El nombre "Python" no proviene del animal, aunque su logo sea una serpiente, sino de la serie cómica británica "Monty Python's Flying Circus", una de las favoritas de Van Rossum.



Evolución

Desde su lanzamiento inicial, Python ha pasado por varias versiones importantes:

- **Python 2.0** (2000): Introdujo la recolección automática de basura y el soporte para Unicode.
- **Python 3.0** (2008): Esta versión trajo cambios significativos que no eran compatibles con Python 2, como una sintaxis más limpia y un enfoque más moderno del manejo de cadenas, entrada/salida, y manejo de errores. Hoy en día, la mayoría de los desarrollos están basados en Python 3.

La versión más actual de Python es la **Python 3.12** (lanzada en octubre de 2023), que continúa mejorando la eficiencia y manteniendo el enfoque en la facilidad de uso, con nuevas características que mejoran el rendimiento y la compatibilidad con las nuevas tecnologías.

La última versión dentro de 3.12 es la 3.12.6 <https://www.python.org/downloads/release/python-3126/>, que básicamente son correcciones de bugs y mejoras técnicas internas.

Usos

Python es simple y ha sido usado para enseñar a programar, pero también es muy versátil y se utiliza en una amplia gama de aplicaciones, tanto en el ámbito académico como profesional.

1. **Desarrollo web:** Frameworks como **Django** y **Flask** han convertido a Python en una opción ideal para construir sitios y aplicaciones web robustas y escalables.
2. **Ciencia de datos y Machine Learning:** Python es el lenguaje líder en este campo, gracias a bibliotecas como **Pandas**, **NumPy**, **Scikit-learn** y **TensorFlow**, que permiten a los científicos de datos manipular grandes volúmenes de datos, crear modelos de aprendizaje automático y realizar análisis avanzados.
3. **Automatización y scripting:** Debido a su simplicidad, Python es ideal para automatizar tareas repetitivas, como el manejo de archivos, la extracción de datos de la web (Web scraping) o la administración de servidores.
4. **Inteligencia artificial y Deep Learning:** Herramientas como **Keras** y **PyTorch** han hecho de Python un estándar para el desarrollo de sistemas de IA, incluyendo redes neuronales, visión artificial y procesamiento de lenguaje natural.
5. **Desarrollo de videojuegos:** Aunque no es el lenguaje principal en la industria de videojuegos, Python se utiliza para prototipar rápidamente juegos, y frameworks como **Pygame** permiten crear juegos sencillos.
6. **Internet of Things (IoT):** Python es utilizado para controlar dispositivos y desarrollar aplicaciones para el Internet de las Cosas (IoT), ya que es compatible con hardware como **Raspberry Pi**. [MicroPython](#) es capaz de ejecutarse en Arduino y ESP32.
7. **Desarrollo de software empresarial:** Gracias a su capacidad para integrarse fácilmente con otros lenguajes y herramientas, Python es utilizado para desarrollar aplicaciones empresariales complejas.

Desarrollos

Empresas que usan Python

Muchas de las empresas más grandes y conocidas del mundo utilizan Python en su infraestructura. Esto se debe a que Python es confiable, flexible y escalable, lo que lo convierte en una herramienta ideal para una variedad de tareas.

- **YouTube:** Usa Python en su infraestructura, especialmente para tareas de análisis de datos.
- **Instagram:** Utiliza Python en su backend para manejar miles de millones de usuarios con una alta eficiencia.
- **Spotify:** Utiliza Python para su análisis de datos y recomendación de canciones, aprovechando las capacidades de Python en análisis de datos masivos.
- **Dropbox:** Toda su infraestructura está basada en Python, y Guido van Rossum también trabajó aquí durante algunos años.
- **NASA:** Utiliza Python en sus simulaciones y análisis científicos debido a la precisión y flexibilidad que proporciona el lenguaje.
- **Reddit:** Uno de los sitios más grandes de discusión en línea, está completamente escrito en Python.
- **Blender:** Un software de creación 3D de código abierto muy popular, utilizado en la industria del cine y los videojuegos.
- **GIMP:** Un editor de imágenes gratuito y de código abierto, comparable a Photoshop.

Python es de los lenguajes más populares entre los desarrolladores profesionales:

<https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language-prof>

Características

Python es un lenguaje de alto nivel de programación **interpretado** y **multiplataforma** cuya filosofía hace hincapié en la **legibilidad** de su código. Se trata de un lenguaje de programación **multiparadigma**, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional.

Filosofía de Python: Principios que fueron descritos por el desarrollador de Python Tim Peters en El Zen de Python:

https://es.wikipedia.org/wiki/Zen_de_Python

Lenguajes compilados o interpretados

Lenguaje compilado: En lenguajes compilados, el código fuente se traduce en código máquina (lo que ejecuta realmente la CPU) por un compilador antes de la ejecución. El programa resultante (binario) es independiente del código fuente y puede ser ejecutado directamente por el sistema operativo. Ejemplos: C, C++.

- **Ventajas:** La ejecución del código suele ser más rápida porque ya está traducido a código máquina.
- **Desventajas:** El proceso de compilación toma tiempo y cualquier cambio en el código fuente requiere recompilar todo el programa. Además, el código compilado es específico para cada plataforma, lo que complica la portabilidad.

Lenguaje interpretado: El código se ejecuta directamente a través de un intérprete que traduce cada línea en tiempo de ejecución. Esto permite una mayor flexibilidad en el desarrollo, ya que no es necesario compilar todo el programa. Python, JavaScript

- **Ventajas:** Mayor facilidad para la depuración, ya que se pueden ejecutar fragmentos de código sin recompilar todo el programa. Es más portable, ya que el mismo código fuente puede ejecutarse en diferentes plataformas.
- **Desventajas:** La ejecución puede ser más lenta que en un lenguaje compilado, ya que el código debe ser interpretado cada vez que se ejecuta.

Tipado en Python

Un lenguaje de programación puede tener tipado dinámico o estático, fuerte o débil.

- **Tipado estático o dinámico:** Define cómo y cuándo se verifica el tipo de dato de una variable.
 - a. **Estático:** El tipo de una variable se declara al definirla y no puede cambiar durante la ejecución. Esto ofrece mayor seguridad pero es más restrictivo. Ejemplos: C, Java.
 - b. **Dinámico:** El tipo de una variable se determina en tiempo de ejecución, según el valor que se le asigne. Esto permite una mayor flexibilidad pero puede introducir errores si no se tiene cuidado. Ejemplo: JavaScript.
- **Tipado fuerte o débil:** Define cómo es de escrito el lenguaje al manejar los tipos de datos en las operaciones.
 - a. **Fuerte:** El sistema de tipos es **estricto**. No se permiten conversiones automáticas de tipos sin intervención explícita, y realizar operaciones con tipos incompatibles genera errores (sumar un número a una cadena). Ejemplos: Java, C++.
 - b. **Débil:** El lenguaje realiza conversiones de tipo implícitas automáticas, lo que puede llevar a resultados inesperados al operar tipos a priori. Ejemplos: PHP, JavaScript.

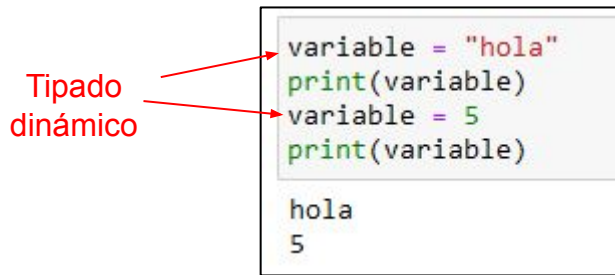
Python tiene tipado dinámico y fuerte.

Dinámico: En Python, las variables no requieren una declaración explícita del tipo de dato. El tipo de cada variable lo asigna el intérprete automáticamente en el momento de recibir un valor y puede cambiar a lo largo del programa.

Tipado
dinámico

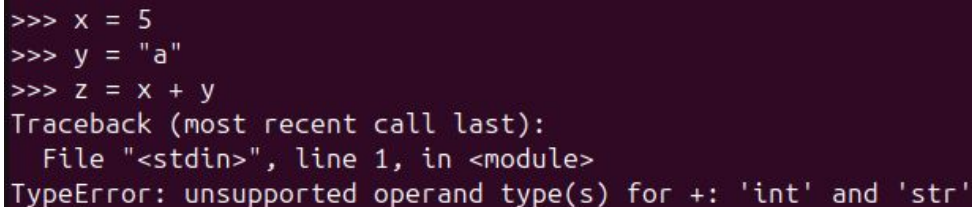
```
variable = "hola"
print(variable)
variable = 5
print(variable)

hola
5
```



Fuerte: Python no permite realizar operaciones que involucren tipos incompatibles sin conversión explícita.

```
>>> x = 5
>>> y = "a"
>>> z = x + y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```



No confundir la sobrecarga de operadores con el tipado débil. En python, `5 * "a"` = `"aaaaa"`.

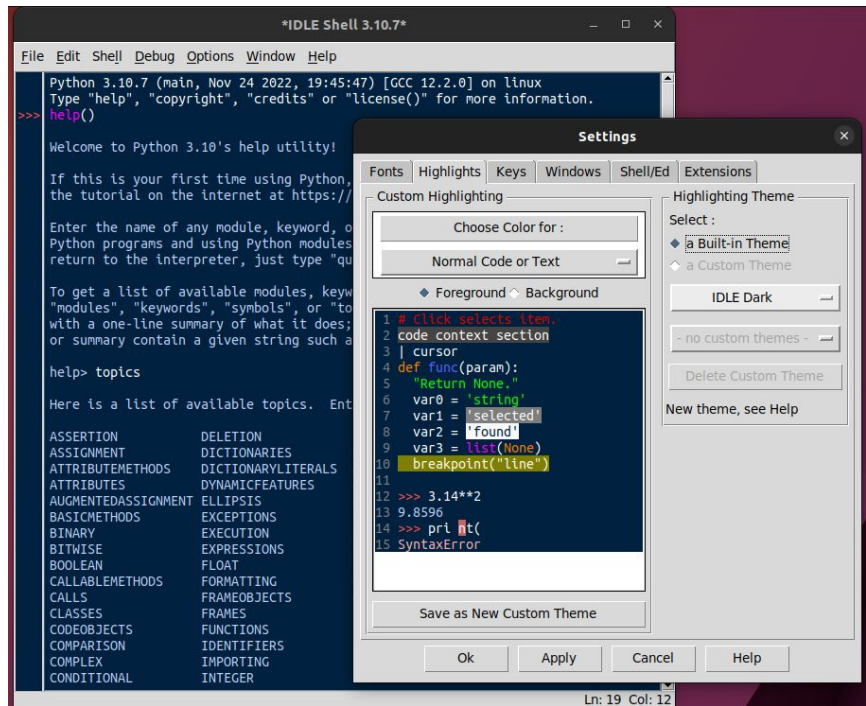
Instalación

Descarga e instalación:

<https://www.python.org/downloads/>

Ejecución en modo interactivo mediante un intérprete de comandos:

- Windows: Buscar en aplicaciones python3.
- Linux: Abrir un terminal y ejecutar “python3”.



El intérprete de Python y la escritura de instrucciones en el shell

- **Interacción con el intérprete:**
 - **Modo interactivo:** Es ideal para probar pequeños fragmentos de código y experimentar con el lenguaje.
 - **Prompts:**
 - `>>>`: Indica que el intérprete está listo para recibir una nueva instrucción.
 - `...:`: Indica que se espera una continuación de la instrucción anterior (e.g., dentro de una definición de función o un bloque condicional). Para salir del bloque, pulsar intro.
 - **Historia de comandos:** Permite volver a ejecutar comandos anteriores utilizando las flechas arriba y abajo.
 - **Salir del intérprete:** `exit()` o `Ctrl+d`
- **Escritura de instrucciones:**
 - **Mayúsculas y minúsculas:** Python distingue entre mayúsculas y minúsculas (case-sensitive).
 - **Espacios en blanco:** La indentación es crucial para definir bloques de código. En Python no existen las llaves `{ }`. Para crear bloques de código basta 1 espacio en blanco, pero se recomienda usar 4 espacios.
 - **Varias instrucciones en 1 línea:** Se pueden escribir, separadas por `;`
 - **Comentarios:**
 - **De una línea:** Comienzan con `#`.
 - **Multilínea:** Se encierran entre triples comillas simples o dobles (`''' comentario '''`)

PEP

PEP (Python Enhancement Proposal o Propuestas de mejoras de Python).

Un PEP es un documento de diseño que proporciona información a la comunidad de Python y describe una nueva característica del lenguaje, de sus procesos o del entorno.

Los PEP son documentos para proponer nuevas funciones, para recopilar opiniones de la comunidad sobre un problema y para documentar las decisiones de diseño que se han aplicado a Python.

El Zen de Python está escrito como la entrada informativa número 20 de las Propuestas de Mejoras de Python.

<https://peps.python.org/pep-0000/>

PEP 8

Son recomendaciones para que el código sea más legible y consistente. Algunas son:

- Usar sangrías de 4 espacios, no tabuladores.
- Recortar las líneas para que no superen los 79 caracteres.
- Usar líneas en blanco para separar funciones y clases y bloques grandes de código.
- Cuando sea posible, poner comentarios en una sola línea, en una línea independiente.
- Usar docstrings: son comentarios de varias líneas para documentar funciones, métodos y clases. Se colocan inmediatamente después de la declaración de una función o clase.
- Usar espacios en expresiones y sentencias:
 - No usar espacios alrededor de paréntesis, corchetes o llaves.
 - Un solo espacio antes y después de los operadores (como =, +, -, *)
- Nombrar las clases y funciones consistentemente:
 - Clases: usar CamelCase
 - Variables, métodos y funciones: snake_case (minúsculas_con_guiones_bajos).
- No usar caracteres no-ASCII en los identificadores.

<https://peps.python.org/pep-0008/>

Sintaxis básica

Indentación (sangría): La sintaxis de Python no utiliza llaves { } para agrupar bloques de código como Java, C o JavaScript. Lo que determina a qué bloque de código pertenece una instrucción es la sangría o indentación. Varias líneas con la misma indentación pertenecen al mismo bloque de código. La imagen derecha genera un error.

```
suma = 0
contador = 1
```

```
suma = 0
contador = 1

Cell In[6], line 2
    contador = 1
    ^
IndentationError: unexpected indent
```

Según el código de buenas prácticas, la **sangría son 4 espacios en blanco**. Algunos IDE configuran automáticamente la tecla tabulador para realizar los 4 espacios.

Sintaxis básica

Comentarios: Los comentarios en una línea se marcan con el carácter #.

Para comentarios en varias líneas (o docstring) utilizamos 3 comillas (simples o dobles)

```
#Comentarios en una línea

print("Hola mundo") #También se pueden situar aquí
...
Comentarios en varias líneas
haciendo uso de 3 caracteres comillas simples
'''

'''
Comentarios en varias líneas
haciendo uso de 3 comillas dobles
'''
print("hola mundo")
```


Sintaxis básica

Variables: Se crean asignando un valor directamente, no es necesario indicar el tipo, ya que el tipo está asociado al valor asignado y puede cambiar durante la ejecución. Se permite la **asignación múltiple**.

- Los nombres pueden contener letras, números o guiones bajos, pero debe empezar por una letra.
- Utilizar nombres descriptivos, a no ser que sea un índice para recorrer vectores o listas (i,j).
- Nombrar mediante snake_case.
- Python es un lenguaje case sensitive, es decir, distingue entre mayúsculas y minúsculas.

Asignación múltiple

```
num_a, num_b = 5, 10  
print(num_a)  
print(num_b)
```

```
5  
10
```

Tipos básicos

- **Numéricos:** Incluyen tipos como **int** (enteros), **float** (números de punto flotante) y **complex** (números complejos).
- **Booleanos:** Representan valores lógicos **True** o **False**.
- **Cadenas de caracteres:** (**str**) pueden utilizarse para representar caracteres individuales o cadenas, usando comillas simples o dobles.
- **None:** Valor especial, similar a null en otros lenguajes. Se retorna desde funciones que no retornan nada explícitamente. Su valor de verdad es falso.

```
# Numéricos
entero = 5
real = 12.2425
complejo = 4 + 5j

# Cadenas
cadena = "Hola Mundo"
cadena2 = 'Hola Mundo'
caracter = 'a'

# Booleanos
mivariable = True
mivariable = False

# None: Representa la ausencia de valor
nada = None
```

Imprimir y leer de teclado

Una **función** es un algoritmo independiente que puede ser invocado, pasándole argumentos, y que nos puede devolver un resultado como consecuencia de su ejecución.

Podemos hacer nuestras propias funciones, pero también existen funciones predefinidas en Python. 2 de ellas son print e input:

- Escribir por pantalla: **print()**. Las cadenas pueden manejarse con comillas simples o dobles. Puede pasarse por parámetro una cadena literal, una variable, o ambas.

```
print("Hola mundo")
```

```
Hola mundo
```

```
mensaje = "hola mundo"
```

```
print(mensaje)
```

```
hola mundo
```

```
variable = 5
```

```
print("Hola mundo, mi número es: ", variable)
```

```
Hola mundo, mi número es: 5
```

Imprimir y leer de teclado

Print interpreta `\n` como salto de línea:

```
print('C:\documentos\nombres')
```

C:\documentos
ombres

Poniendo una `r` antes de abrir comillas no interpreta caracteres especiales:

```
print(r'C:\documentos\nombres')
```

C:\documentos\nombres

Imprimir en varias líneas utilizando 3 comillas:

```
print("""\nUSO: comando [OPTIONS]\n    -h\n    -H hostname\n""")
```

USO: comando [OPTIONS]
-h
-H hostname

Muestra este mensaje
Hostname a conectar

Muestra este mensaje
Hostname a conectar

Imprimir y leer de teclado

- Leer datos de teclado: **input()**

```
num1 = input()
print("El número es:", num1)

7
El número es: 7
```

Podemos mostrar un mensaje cuando se invoca input().

```
num1 = input("Introduce un número: ")
print("El número es:", num1)
```

Introduce un número:

Imprimir y leer de teclado

`input()` devuelve una cadena de caracteres. La función **`type()`** muestra el tipo de la variable.

```
num1 = input("Introduce un número: ")  
type(num1)
```

Introduce un número: 7

str

Si queremos tratarlo como un entero hay que realizar “**casting**” (convertir de un tipo a otro)

```
num1 = int(input("Introduce un número: "))  
type(num1)
```

Introduce un número: 7

int

Operadores aritméticos y de asignación

```
# Operadores aritméticos
suma = 5 + 2
resta = 5 - 2
multiplicacion = 5 * 2
division = 5 / 2
division_entera = 5 // 2
modulo = 5 % 2
potencia = 5 ** 2
```

```
# Operadores de asignación y operación
asignacion = 5
suma += 2
resta -= 2
multiplicacion *= 2
division /= 2
division_entera //= 2
modulo %= 2
potencia **= 2
```

Precedencia: Los operadores siguen un orden de evaluación específico:
Paréntesis, Exponenciación, Multiplicación, División, Suma, Resta (PEMDAS)

Operadores de comparación y lógicos

```
# Operadores de comparación
resultado = a == b
resultado = a != b
resultado = a > b
resultado = a < b
resultado = a >= b
resultado = a <= b
```

```
# Operadores lógicos
resultado = var1 and var2
resultado = var1 or var2
resultado = not var1
```


Operadores de identidad

Objetos

- Python todo es un objeto, incluso una variable con un valor entero.
- Como objeto, tiene su **id** de referencia en memoria que le identifica.
- La función **id()** permite saber el identificador de un objeto.

```
entero = 5  
print(id(entero)) # Imprime 11754024
```

- Los operadores **is** / **is not** permiten comprobar si dos variables apuntan al mismo objeto.

```
a = 5  
b = a  
print(a is b) # Imprime True  
print(a is not b) # Imprime False
```

Identidad no es igualdad

Los operadores de identidad comprueban si 2 variables apuntan al mismo identificador de objeto (dirección de memoria).

En Python, cuando creas varias variables con el mismo valor de tipos básicos, pueden apuntar al mismo objeto. Esto sucede debido a un proceso llamado **interning** o **caching**, en el cual Python reutiliza objetos con el mismo tipo y valor para ahorrar memoria y mejorar el rendimiento.

```
a = 5
print(id(a)) # 11754024
b = 4
print(id(b)) # 11753992
b += 1
print(id(b)) # 11754024

print(a == b) # True
print(a is b) # True
```

Mismo **id** cuando tienen el mismo valor



con números mayores
no se comporta igual

```
a = 555
print(id(a)) # 140221606903440
b = 554
print(id(b)) # 140221606895504
b += 1
print(id(b)) # 140221606903728

print(a == b) # True
print(a is b) # False
```

Operadores de pertenencia

Los operadores **in**, **not in** comprueban si un valor (literal o variable) se encuentra dentro de un conjunto, como una lista, string, etc.

```
cadena = "Hola Mundo"  
resultado = "Hola" in cadena # True  
resultado = "Hola" not in cadena # False
```

Operadores a nivel de bits

En ciertas tareas de programación, como programación de microcontroladores, interfaces hardware, direcciones IP o criptografía, donde **cada bit tiene un significado específico**, estos operadores te permiten activar, desactivar, o consultar el valor de bits individuales.

- Podemos almacenar un número binario en una variable anteponiendo '0b'
- Para consultar el valor binario de una variable, podemos usar la función `bin()`. De lo contrario, se mostrará su valor decimal. Los 0 a la izquierda no se visualizan.

```
numero_binario = 0b0101
print(bin(numero_binario)) # Salida -> 0b101
```

- Otra opción es usar la función `format(número, '0nb')`. Siendo n el número de bits.

```
numero_binario = 0b0101
print(format(numero_binario, '08b')) # Salida -> 00000101
```

Operadores a nivel de bits

Operador	Nombre	Ejemplo	Descripción
&	AND	a & b	AND: Devuelve 1 si ambos bits son 1.
	OR	a b	OR: Devuelve 1 si cualquier bit es 1.
^	XOR	a ^ b	XOR: Devuelve 1 si los bits son diferentes.
~	NOT	~a	Cambia 1 por 0 y viceversa. (incluyendo “bit signo”, al imprimir, python interpreta como complemento a 2).
<<	Desp. izquierda	a << n	Desplaza a la izquierda n bits, metiendo ceros
>>	Desp. derecha	a >> n	Desplaza a la derecha n bits, metiendo ceros

Operador Walrus :=

Este operador asigna y devuelve el contenido de una variable.

Sin walrus

```
cadena = "Python"  
print(cadena)
```

Con walrus

```
print(cadena := "Python") # Salida -> Python
```

type()

- **type()** es una función predefinida que nos dice la clase de objeto de una variable o literal.

```
entero = 5  
print("Tipo: ", type(entero))
```

Tipo: <class 'int'>

```
real = 15.2  
print("Tipo: ", type(real))
```

Tipo: <class 'float'>

```
cadena = "Python"  
print("Tipo: ", type(cadena))
```

Tipo: <class 'str'>

Constructores

Podemos construir un objeto a través de otro valor mediante los constructores.

- `str()`
- `int()`
- `float()`
- `bool()`

```
cadena = str(5)
print(cadena)
print("Tipo: ", type(cadena))
```

```
5
Tipo:  <class 'str'>
```

```
entero = int("5")
print(entero)
print("Tipo: ", type(entero))
```

```
5
Tipo:  <class 'int'>
```

```
real = float(5)
print(real)
print("Tipo: ", type(real))
```

```
5.0
Tipo:  <class 'float'>
```


Valores True o False

Valores resultantes al evaluar, mediante un if o while, o conversión a booleano con bool(), un valor distinto a booleano.

- Son evaluados a **False**:
 - El 0 y el 0.0
 - Una estructura vacía (por ejemplo una lista)
 - Una cadena vacía ""
 - El valor None
- Son evaluados a **True**:
 - Una estructura con contenido
 - Una cadena con al menos un carácter
 - Un numérico distinto de 0

Excepciones comunes

Python no es un lenguaje compilado, no tendremos al compilador avisando de errores, sino que estos se darán al ejecutar el código. Cuando hay un error incompatible con la ejecución, el intérprete lanzará una excepción de forma automática. Es importante conocer de antemano las más usuales para saber por qué falla nuestro código.

- **SyntaxError**: Error de sintaxis en el código. Por ejemplo, falta cerrar un paréntesis.
- **IndentationError**: Indentación del código es incorrecta.
- **NameError**: Se intenta usar una variable o función no definida.
- **TypeError**: Se intenta realizar una operación sobre tipos no compatible, por ejemplo `5 + "hola"`
- **ValueError**: Se lanza cuando una función recibe un valor incorrecto, por ejemplo `int("hola")`
- **ZeroDivisionError**: Se lanza cuando se está intentando dividir por 0.
- **IndexError**: Se lanza cuando se intenta acceder a índice que está fuera de los límites.

Creación de scripts y ejecución

Crearemos nuestros archivos de código utilizando un **editor de código**, con la función de resaltado de sintaxis como mínimo. Podemos utilizar bloc de notas, nano, vim u otras herramientas como un IDE (VSCode, Pycharm). Los archivos de Python tienen **extensión .py**

- **Shebang:** Nuestro archivo de código fuente principal (donde estará el “main”) debería tener en la primera línea el [Shebang](#). Esto indica al sistema operativo qué intérprete debe invocar para ejecutar el script. La línea contiene los símbolos ‘#!’ seguido de la ruta donde se encuentra el intérprete. En Linux puede ser `#!/bin/python3` o `#!/usr/bin/python3`.
- **Ejecución:** Podemos ejecutar el script invocando al intérprete mediante `python3 archivo.py`. O podemos darle al archivo permisos de ejecución y ejecutarlo directamente con `./archivo.py`
- **Permisos en Linux:** Para dar permisos de ejecución a un archivo -> `chmod u+x archivo.py`

Módulos

Un **módulo** es un fichero .py que **alberga un conjunto de funciones, variables o clases** y que puede ser usado por otros módulos. Nos permiten reutilizar código y organizarlo mejor en ficheros independientes.

Por ejemplo, podemos definir un módulo **sumres.py** con dos funciones **suma()** y **resta()**.

Dicho módulo puede ser **importado** en un fichero para usar las funciones. Tenemos varias formas de hacerlo:

- **import modulo** “Importamos todo el módulo”
 - Usamos las funciones a través del nombre del módulo-> **sumres.suma(3, 4)**
- **Renombrando con as** -> “Igual que el anterior pero renombrando al módulo”
 - **import sumres as sr**. Usamos las funciones con el nuevo nombre -> **sr.suma(3, 4)**
- **from modulo import lista** “Importamos solamente los elementos indicados en la lista”
 - **from sumres import suma, resta**
 - Usamos las funciones directamente, sin nombrar al módulo -> **suma(3, 4)**
- **from modulo import *** “Importamos todo el módulo”
 - Usamos las funciones directamente, sin nombrar al módulo -> **suma(3, 4)**

A tener en cuenta: si definimos funciones o variables con el mismo nombre, prevalecen las del fichero actual.

VSCode

- **VSCode** es un **IDE**, Entorno de desarrollo integrado (en inglés integrated development environment). Este IDE nos permite codificar, depurar, probar y ejecutar el programa.
- Otro entorno muy usado en Python es PyCharm.
- Para Análisis de Datos es recomendable usar Jupyter Notebook y/o Anaconda.
- Plugins básicos para empezar con VSCode: Python (incluye Pylance y Python Debugger). Además, con el Plugin correspondiente, podemos abrir archivos de Jupyter Notebook.

Entornos virtuales

¿Qué es un entorno virtual?

Un **entorno virtual** es un entorno aislado dentro de tu sistema donde puedes instalar una versión específica de Python y librerías de terceros sin interferir con otras instalaciones de Python en tu máquina. Esto es útil para evitar conflictos entre proyectos que puedan requerir diferentes versiones de bibliotecas o dependencias. Con entornos virtuales, diferentes aplicaciones pueden usar entornos de ejecución diferentes.

¿Por qué usar entornos virtuales?

- **Aislamiento:** Cada proyecto puede tener sus propias dependencias sin afectar otros proyectos.
- **Evitar conflictos:** Si dos proyectos necesitan diferentes versiones de la misma biblioteca, el entorno virtual resuelve este problema.
- **Organización:** Mantiene los paquetes y dependencias de cada proyecto separados, facilitando su administración.
- **Portabilidad:** Facilita compartir proyectos, ya que puedes instalar las mismas dependencias en otro entorno fácilmente.

El entorno virtual es un **directorío** que contiene una **instalación de Python de una versión en particular**, además de unos cuantos **paquetes/módulos adicionales** que instalaremos manualmente según las necesidades del proyecto.

Entornos virtuales

- [venv](#) instalará en el entorno virtual la versión de Python desde la que se ejecuta el comando de creación. Es decir, si se ejecuta el comando desde una instalación de Python3.12, se instalará la versión 3.12 dentro del entorno virtual. Para saber la versión de Python instalada ejecutamos **"python --version"**
- Para **crear** un entorno virtual, decide en qué directorio quieres crearlo (normalmente en el directorio del proyecto) y ejecuta [venv](#) indicando la ruta del directorio: **"python -m venv /ruta_proyecto/mi-env"**
- Esto creará el directorio **mi-env** si no existe, y también creará directorios dentro de él que contienen una copia del intérprete de Python y varios archivos de soporte.
- Una vez creado el entorno virtual se debe **activar**, ejecutando:
 - En Windows: **ruta_proyecto\mi-env\Scripts\activate**
 - En Linux: **source ruta_proyecto/mi-env/bin/activate** Este script es para bash. Si usas **csh**, ejecuta **activate.csh**
- Activar el entorno virtual cambiará el prompt de tu consola para mostrar que entorno virtual está usando, y modificará el entorno para que al ejecutar python sea con esa versión e instalación en particular.
- Para desactivar el entorno virtual ejecutar el comando: **deactivate**
- Para eliminar un entorno simplemente borrar el directorio.

Gestor de paquetes

La instalación de Python viene con varios módulos/librerías por defecto, lo que se denomina la librería estandar. Pero gran parte del potencial de Python se debe a la gran cantidad de librerías de terceros como Pandas, NumPy, Matplotlib, etc. Estas librerías hay que descargarlas manualmente en nuestro entorno virtual.

¿**Qué es pip?** pip es el **gestor de paquetes** de Python que te permite instalar, actualizar y gestionar bibliotecas y módulos de terceros que están disponibles en el **Python Package Index** ([PyPI](#)).

- Verificar la instalación de pip: `pip --version`
- Los paquetes se instalan en el entorno virtual activo y no en otros ni en el directorio global de Python.
- Instalar paquetes, ejecuta: `pip install nombre_del_paquete`
- Ver los paquetes instalados: `pip list`
- Actualizar la versión de un paquete: `pip install --upgrade nombre_del_paquete`
- Desinstalar un paquete: `pip uninstall nombre_del_paquete`

Archivo de requerimientos

Cuando trabajas en un proyecto, puedes generar un archivo `requirements.txt` que enumere todas las dependencias de tu proyecto, esto es, una lista de módulos o paquetes necesarios. Esto es útil para compartir el proyecto con otras personas o para recrear el entorno en otra máquina de forma rápida.

- Crear el archivo de texto plano con los nombres de los paquetes en forma de lista, uno debajo de otro.
- Para instalar los paquetes del archivo: `pip install -r requirements.txt`

Otros comandos:

- Verificar información de un paquete: `pip show nombre_del_paquete`
- Buscar un paquete en PyPI: ~~`pip search nombre_del_paquete`~~ Buscar en <https://pypi.org/>

Entorno virtual para los módulo Análisis de Datos

No es estrictamente necesario hacerlo así, pero para seguir una guía común:

- Crear un directorio de nombre proyectos y entrar en él:
 - `mkdir proyectos & cd proyectos`
- Crear el entorno virtual oculto de nombre “.adatos-env”
 - `python3 -m venv .adatos-env`
- Activar el entorno virtual
 - `source .adatos-env/bin/activate`
- Instalar las librerías para ejecutar documentos de Jupyter Notebook:
 - `pip install ipykernel`
- Abrir VSCode en el directorio proyectos
 - `code .`
- Instalar extensiones: Jupyter (es un pack con varias) Python (instala además Pylance y Debugger)
- Los archivos de Jupyter Notebook se crean con la extensión “.ipynb”
- En VSCode antes de ejecutar nos pedirá seleccionar un kernel de python. Seleccionamos el entorno que acabamos de crear.
- Los archivos normales de python tienen extensión “.py”
- Para instalar nuevas librerías necesitamos activar el entorno antes de ejecutar `pip install...` pero para ejecutar un programa con ese entorno no es necesario, basta con seleccionarlo en VSCode.