

MANUAL 101

Introducción a la Programación con Python

(la maga
de Python,)

Magalí Dominguez Lalli

Índice

Unidad I: ALGORITMIA Y PROGRAMACIÓN	2
Unidad II: ¿QUÉ ES PYTHON?	7
Unidad III: SINTAXIS, COMENTARIOS Y BUENAS PRÁCTICAS	15
Unidad IV: DATOS, VARIABLES Y OPERADORES	19
Unidad V: ESTRUCTURAS DE CONTROL DE FLUJO	35
Unidad VI: FUNCIONES, MÉTODOS, FUNCIONES PREDEFINIDAS Y COMPRESIÓN DE LISTAS	40
Unidad VII : CLASES, OBJETOS Y POO	54
Unidad VIII: ARCHIVOS, MÓDULOS Y PAQUETES	61
Unidad IX: EXCEPCIONES	70

Unidad I

ALGORITMIA Y PROGRAMACIÓN

¿Qué es un algoritmo?

Un algoritmo es un *conjunto ordenado y finito de operaciones que permite encontrar la solución a un problema cualquiera*. Entonces, una receta de cocina, por ejemplo, es un algoritmo.

ETAPAS

1. Definición del problema
2. Análisis del problema
3. Diseño y desarrollo del algoritmo
4. Prueba de escritorio y depuración
5. Documentación

CARACTERÍSTICAS



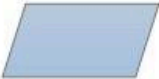


- **Preciso**: no puede ser ambiguo.
- **Definido**: todas las veces que se siga el mismo algoritmo, debe obtenerse el mismo resultado.
- **Finito**: debe tener un inicio y un fin.

TIPOS

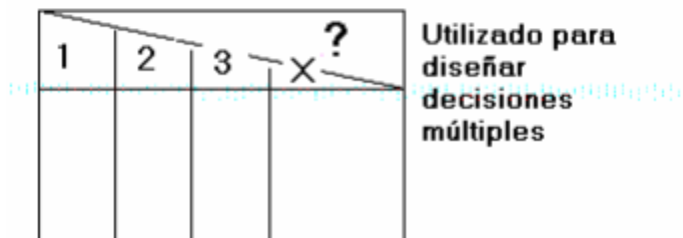
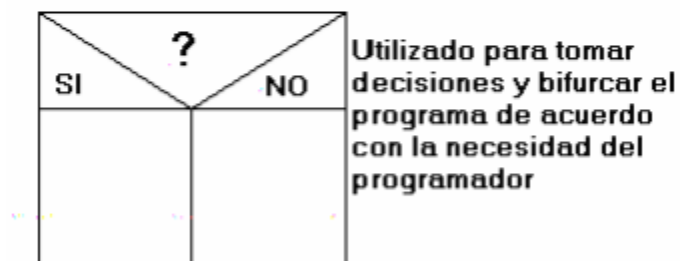
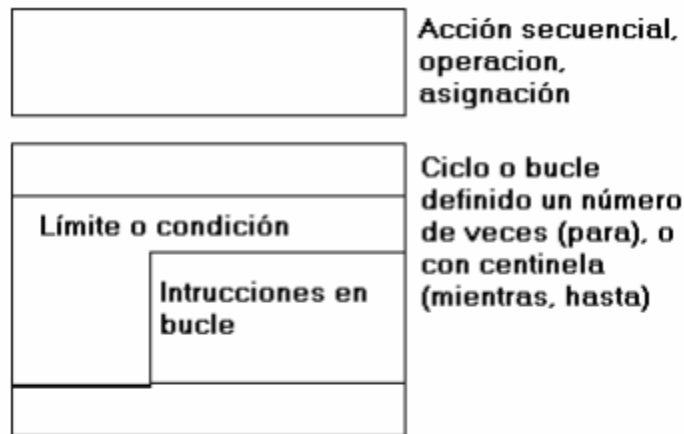
- **Cualitativo**: todo algoritmo que no incluye operaciones aritméticas en ninguno de sus pasos.
- **Cuantitativo**: todo algoritmo que incluye operaciones aritméticas en al menos uno de sus pasos.

TÉCNICAS DE REPRESENTACIÓN

1. **Diagramas de flujo**: este tipo de representación gráfica emplea una serie determinada de figuras geométricas que representan cada paso puntual del proceso que está siendo evaluado. Estas formas, definidas de antemano, se conectan entre sí a través de flechas y líneas que marcan la dirección del flujo y establecen el recorrido del proceso, como si fuera un mapa. Son un mecanismo de control y descripción de procesos, que permiten una mayor organización, evaluación o replanteamiento de secuencias de actividades y procesos de distinta índole, dado que son versátiles y sencillos.

Símbolo	Nombre	Función
	Inicio / Final	Representa el inicio y el final de un proceso
	Línea de Flujo	Indica el orden de la ejecución de las operaciones. La flecha indica la siguiente instrucción.
	Entrada / Salida	Representa la lectura de datos en la entrada y la impresión de datos en la salida
	Proceso	Representa cualquier tipo de operación
	Decisión	Nos permite analizar una situación, con base en los valores verdadero y falso

2. **Diagramas de Nassi-Scheideman**: también conocidos como diagramas de Chapin, corresponden a un tipo de diagramación estructurada. Las acciones se escriben en rectángulos o cajas sucesivas, donde cada caja puede tener más de una acción.



3. **Pseudocódigo**: esta técnica permite escribir el algoritmo mediante palabras normales de una lengua en particular (por ejemplo, español), utilizando verbos en imperativos: inicie, lea, escriba, sume, etc.

```
1  Proceso digitos
2      Definir x Como Entero;
3      Para x<-0 Hasta 9 Con Paso 1 Hacer
4          Escribir x;
5      FinPara
6
7  FinProceso
8
```

¿Qué es un programa?

Un programa es una *secuencia lógica de instrucciones mediante las cuales se ejecutan diferentes acciones de acuerdo con los datos que se estén procesando*. Es decir, un programa es un algoritmo o una secuencia de algoritmos. La diferencia entre un algoritmo y un programa es que, si bien ambos hacen referencia a una serie de instrucciones, los algoritmos pueden estar escritos en código o en lenguaje natural, mientras que los programas sólo pueden estar escritos en *lenguaje de programación*.

¿Qué es la programación?

La programación es el *proceso de transformar un método para resolver problemas en uno que pueda ser entendido por la computadora*.

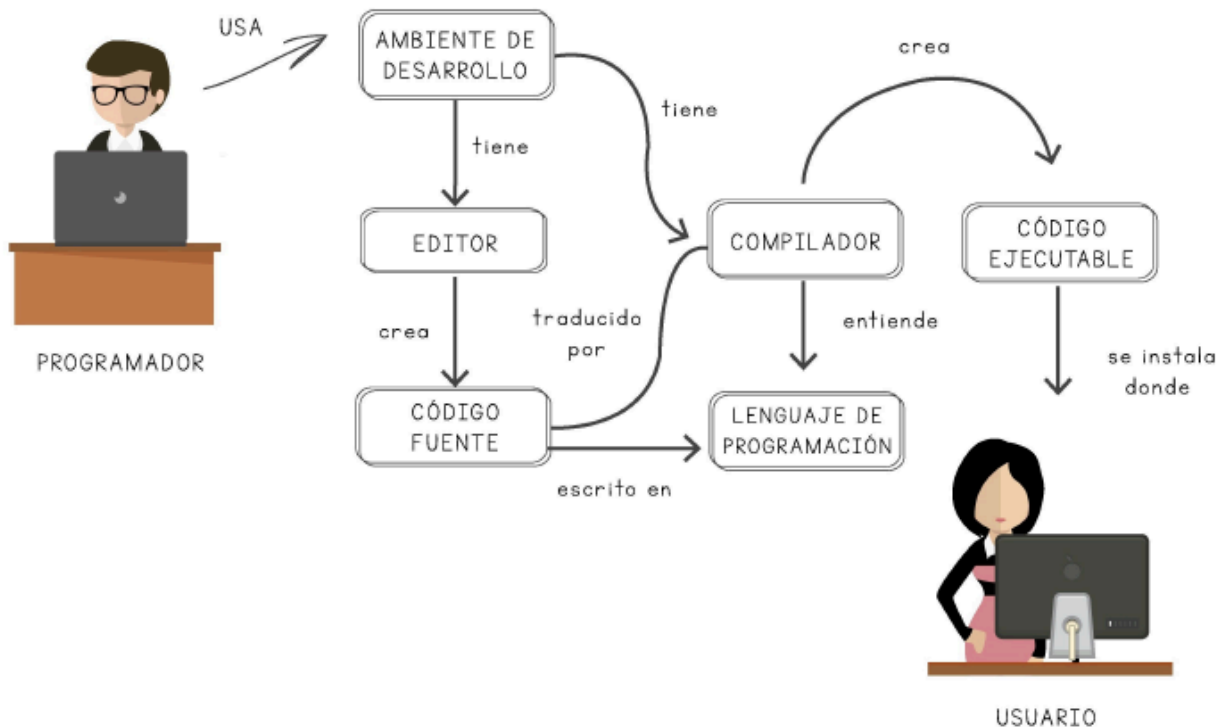
¿Qué es un lenguaje de programación?

Un lenguaje de programación es *un sistema de símbolos y reglas que permite la construcción de programas con los que la computadora puede operar así como resolver problemas de manera eficaz*. Éstos contienen un conjunto de instrucciones que nos permiten realizar operaciones de entrada / salida, cálculo, manipulación de textos, lógica / comparación y almacenamiento / recuperación.

CLASIFICACIÓN

- **Lenguaje Máquina:** *aquellos cuyas instrucciones son directamente entendibles por la computadora y no necesitan traducción posterior para que la CPU pueda comprender y ejecutar el programa.* Las instrucciones en lenguaje máquina se expresan en términos de la unidad de memoria más pequeña, el bit (dígito binario 0 ó 1).
- **Lenguaje de Bajo Nivel** (ensamblador): *en este tipo, las instrucciones se escriben en códigos alfabéticos conocidos como mnemotécnicos para las operaciones y direcciones simbólicas.*
- **Lenguaje de Alto Nivel:** los lenguajes de programación de alto nivel (BASIC, pascal, cobol, fortran, etc.) *son aquellos en los que las instrucciones o sentencias a la computadora son escritas con palabras similares a los lenguajes humanos (en general en inglés), lo que facilita la escritura y comprensión del programa.*

ENTONCES...



Unidad II

¿QUÉ ES PYTHON?

Python es un *lenguaje de programación* creado por Guido van Rossum a principios de los años 90, cuyo nombre está inspirado en el grupo de cómicos ingleses “*Monty Python*”. Es un lenguaje similar a Perl, pero con una *sintaxis muy limpia que favorece un código legible*.

Se trata de un lenguaje interpretado o de script, con tipado dinámico, fuertemente tipado, multiplataforma, multiparadigma y de código abierto.

LENGUAJE INTERPRETADO O DE SCRIPT

Esto quiere decir que se ejecuta utilizando un programa como intermediario llamado *intérprete*, en lugar de compilar el código a lenguaje máquina que pueda comprender y ejecutar directamente una computadora (como lo hacen los lenguajes compilados). La ventaja de los lenguajes compilados es que su ejecución es más rápida, pero los lenguajes interpretados son más flexibles y portables.

Sin embargo, Python posee muchas características de los lenguajes compilados, por lo que mucha bibliografía dice que es semi-interpretado. En Python, como en Java y muchos otros lenguajes, el código fuente se traduce a un pseudo-código máquina intermedio llamado bytecode la primera vez que se ejecuta, generando archivos .pyc o .pyo (bytecode optimizado), que son los que se ejecutarán en sucesivas ocasiones.

TIPADO DINÁMICO

Se refiere a que *no es necesario declarar el tipo de dato que va a contener una determinada variable* en su declaración, sino que *éste se determinará en tiempo de ejecución* según el tipo del valor al que se asigne, *pudiendo cambiar tantas veces como sea necesario*.



```
numero = 5
print(numero)

numero = 'diez'
print(numero)

>>> 5
>>> diez
```

FUERTEMENTE TIPADO

No se permite tratar a una variable como si fuera de otro tipo, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente.

```
numero = 10
numero_2 = "cinco"

print(numero + numero_2)

>>> Traceback (most recent call last)
Cell In[1], line 4
      1 numero = 10
      2 numero_2 = "cinco"
----> 4 print(numero + numero_2)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

MULTIPLATAFORMA

El intérprete de Python está disponible variadas plataformas (UNIX, Solaris, Linux, DOS, Windows, OS/2, Mac OS, etc.), entonces, si no utilizamos librerías específicas de cada plataforma o S.O., nuestro programa podrá correr en todos estos sistemas sin cambios significativos.

MULTIPARADIGMA

Con Python se puede trabajar con múltiples paradigmas: programación imperativa, programación funcional, programación orientada a aspectos y programación orientada a objetos. Este último es, tal vez, el más utilizado.

La orientación a objetos es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa, y la ejecución del mismo consiste en una serie de interacciones entre los objetos.

DE CÓDIGO ABIERTO

Por lo tanto, es de *libre distribución*. Cualquiera con una computadora, puede descargarlo, utilizarlo y acceder a su documentación; al igual que cualquiera de sus librerías distribuibles por TPSF.

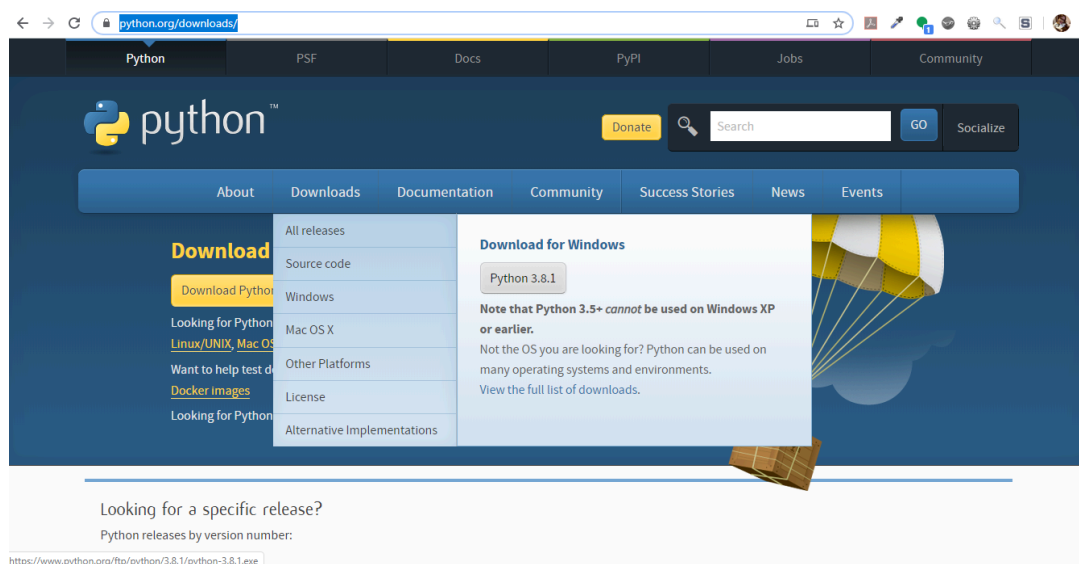
Descargar Python

Python se descarga desde la página oficial <https://www.python.org/downloads/>. La versión actual es la 3.12.3, siempre se recomienda trabajar con la última por una cuestión de costumbre a futuro, pero pueden trabajar con la que deseen. Python 2 ya no tiene soporte, así que lo mejor es que utilicen del 3.x.x en adelante.

Como pueden observar, si clickean en la solapa de Downloads, la web de Python reconocerá qué S.O. utilizan y les ofrecerá descargar la última versión para dicho S.O.. Recuerden que si utilizan Windows, antes de descargar Python, tienen que verificar si se ejecutará en Windows 32bits o 64bits. Para hacerlo deben verificarlo en "Tipo de sistema" en la sección de "Acerca de". Para llegar ahí pueden hacerlo por uno de estos métodos:

- Presionar la *tecla de Windows* y la tecla *Pause/Break* al mismo tiempo
- Abrir el *Panel de Control* desde el *menú de Windows*, después acceder a *Sistema & Seguridad*, luego a *Sistema*
- Presionar el botón de Windows, luego acceder a *Configuración > Sistema > Acerca de*

Más abajo en el sitio, si scrollean, pueden descargarse versiones anteriores.



Looking for a specific release?
Python releases by version number:

Release version	Release date		Click for more
Python 3.8.1	Dec. 18, 2019	Download	Release Notes
Python 3.7.6	Dec. 18, 2019	Download	Release Notes
Python 3.6.10	Dec. 18, 2019	Download	Release Notes
Python 3.5.9	Nov. 2, 2019	Download	Release Notes
Python 3.5.8	Oct. 29, 2019	Download	Release Notes
Python 2.7.17	Oct. 19, 2019	Download	Release Notes
Python 3.7.5	Oct. 15, 2019	Download	Release Notes
Python 3.8.0	Oct. 14, 2019	Download	Release Notes

[View older releases](#)

Licenses

All Python releases are [Open Source](#).
Historically, most, but not all,
Python releases have also been GPL-

Sources

For most Unix systems, you must
download and compile the source
code. The same source code archive

Alternative Implementations

This site hosts the "traditional"
implementation of Python

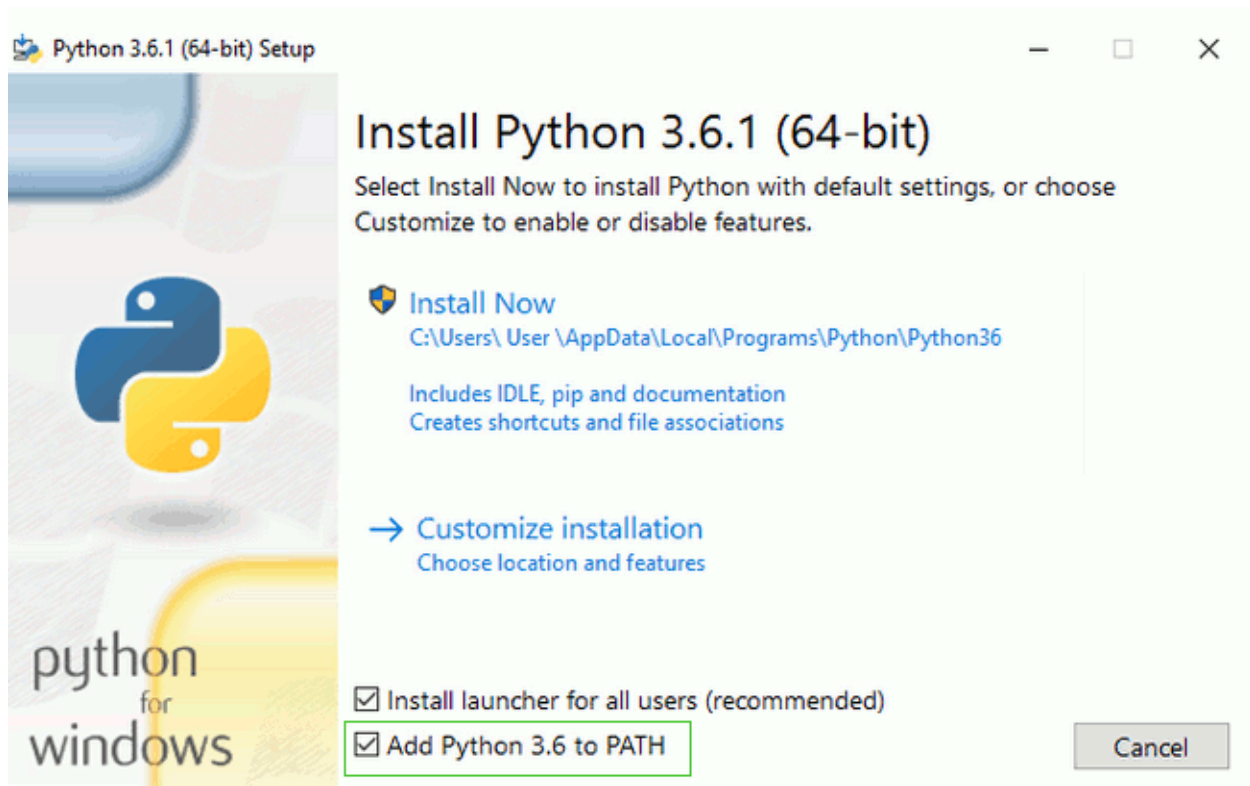
History

Python was created in the early
1990s by Guido van Rossum at
Stichting Mathematisch Centrum in

Una vez descargado, hay que instalarlo, y eso varía según el S.O. en el que se trabaje.

WINDOWS

Pueden proceder a ejecutarlo e instalarlo. Es importante dar click en los check box "[Add Python <versión> to PATH](#)" o "[Add Python to your environment variables](#)" y por último hacer click en "[Install Now](#)", como se muestra aquí:



Cuando la instalación se complete, observarán un cuadro de diálogo con un enlace que pueden clicar para saber más sobre Python o sobre la versión que han instalado.

MacOS

Antes de instalar Python en OS X, deben asegurarse de que la configuración de su Mac permite instalar paquetes que no estén en la App Store. Esto lo pueden chequear clickeando en *preferencias del sistema* (System Preferences, está en la carpeta Aplicaciones), dar click en *Seguridad y privacidad* (Security & Privacy) y luego la pestaña *General*. Si tienen "Permitir aplicaciones descargadas desde:" (Allow apps downloaded from:) establecida a "Mac App Store," cambiar a "Mac App Store y desarrolladores identificados" (Mac App Store and identified developers). Luego:

- Descargar el archivo Mac OS X 64-bit/32-bit installer
- Dar doble click en python-3.x.x-macosx10.6.pkg para ejecutar el instalador

LINUX

Es muy posible que ya tengan instalado Python de serie. Para verificar si es así (y qué versión es), abran una consola (emulador del sistema) y escriban el siguiente comando:

```
$ python3 --version
```

Si tienen instalada al menos la versión 3.4.0, entonces no tienen que actualizar. Si no lo tienen instalado, o si quieren una versión diferente, primero verifiquen qué distribución de Linux están usando con el siguiente comando:

```
$ grep ^NOMBRE= /etc/os-release
```

Después, dependiendo del resultado, deben seguir en la consola y escribir lo siguiente:

```
$ sudo apt install python3
```

Con cualquiera de las tres opciones, verificar si realmente se instaló con el siguiente comando en la consola:

```
$ python3 --version
```

Les devolverá la versión instalada una vez que de enter:

```
Python 3.6.1
```

Codear Python

Para poder codear (acción de escribir código) en Python, se necesitan algunos elementos que, muchas veces, dependen de los gustos, costumbres y/o comodidades de cada uno.

Por un lado están los IDEs (Integrated Development Environments) que son *Entornos de Desarrollo Integrado* (algunas bibliografías los llaman Entornos de Desarrollo Interactivo). Proporcionan servicios integrales para el desarrollo de software.

IDEs

- **PyDev** de Eclipse: <https://www.pydev.org/download.html>
- **PyCharm** de JetBrains: <https://www.jetbrains.com/es-es/pycharm/download/> (Community)
- **Vim**: <https://www.vim.org/download.php>
- **Spyder**: para empezar, si quieren descargar el paquete *Anaconda* y tienen las dos opciones, tanto Spyder como *Jupyter Notebook* (se utiliza mucho para data science). Descargando Anaconda, no solo se descargan esos IDEs, también el lenguaje y varios paquetes útiles para utilizar a futuro. Dato de color: también sirve para codear R.
<https://www.anaconda.com/distribution/>

Por otro lado están los editores de texto, que son eso, editores de texto a los que se le pueden agregar diferentes plugins para que funcionen similar a los IDE.

EDITORES DE TEXTO

- **Sublime Text**: <https://www.sublimetext.com/3>
- **Atom**: <https://atom.io/>
- **Visual Studio Code**: <https://code.visualstudio.com/download>

Tanto los IDEs como los Editores de Texto mencionados están disponibles para todas las plataformas. En caso de no tener espacio en disco, pueden usar google colab que funciona igual que Jupyter (por celdas):

<https://colab.research.google.com/notebooks/intro.ipynb#recent=true>

PIP

Al igual que cualquier otro lenguaje de programación, Python admite librerías, módulos y paquetes de terceros que se pueden instalar y utilizar para funciones especiales o simplemente para reducir código. Pueden encontrarlos en un repositorio central llamado **PyPI** (Python Package Index) <https://pypi.org/>.

Descargar, instalar y administrar estos paquetes a mano puede llevar mucho tiempo, por lo tanto, muchos desarrolladores de Python confían en una herramienta especial llamada *pip* para que Python haga todo mucho más fácil y

rápido.

pip no es más que *Paquetes de Instalación*. Es una utilidad de línea de comandos que les permite instalar, reinstalar o desinstalar paquetes PyPI con un comando simple y directo: *pip*.

Desde la versión 2.7 de Python, *pip* ya viene con ella, las versiones anteriores requieren su descarga e instalación por separado.

Siempre que quieran instalar un paquete nuevo, desde su consola de comandos deben ejecutar el siguiente:

```
pip install <nombre_paquete>
```

ACTUALIZAR PIP

Desde cualquiera de los tres S.O. se hace desde la Terminal o consola de comandos.

Windows

```
python -m pip install -U pip
```

OS

```
pip install -U pip
```

Linux

```
pip install -U pip
```

Unidad III

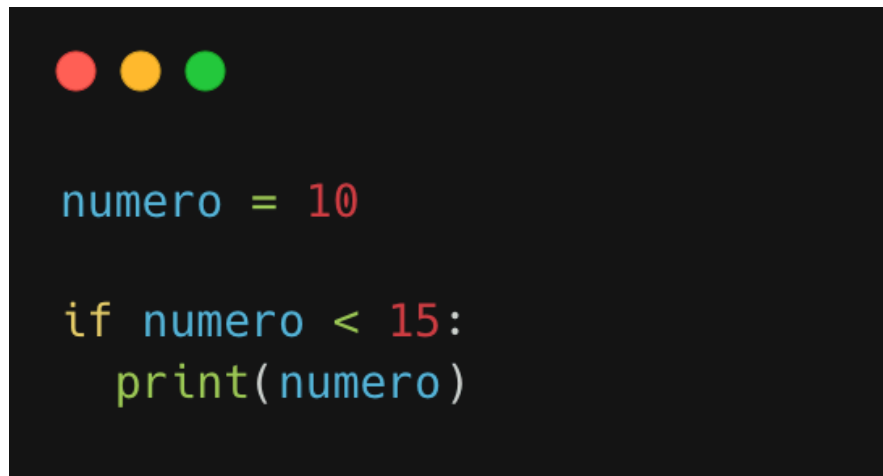
SINTAXIS, COMENTARIOS Y BUENAS PRÁCTICAS

Sintaxis

Es importante tener en cuenta que *la clave de la sintaxis de python es la*

indentación. A diferencia de otros lenguajes, no se utilizan ; para indicar el fin de una línea de código, ni {} para asociar líneas de código en bloque. Por esto es que se dice que su código es limpio y fácil de leer.

La indentación son cuatro espacios después del comienzo, *es una sangría, que nos indicará a qué parte del código pertenece una instrucción o bloque.*

A screenshot of a code editor with a dark background. At the top left, there are three colored circles: red, yellow, and green. Below them, the following Python code is displayed with syntax highlighting: 'numero = 10' on one line, and an indented block 'if numero < 15:' followed by 'print(numero)' on the next line, with four spaces between the 'if' and the opening colon.

```
numero = 10

if numero < 15:
    print(numero)
```

Comentarios

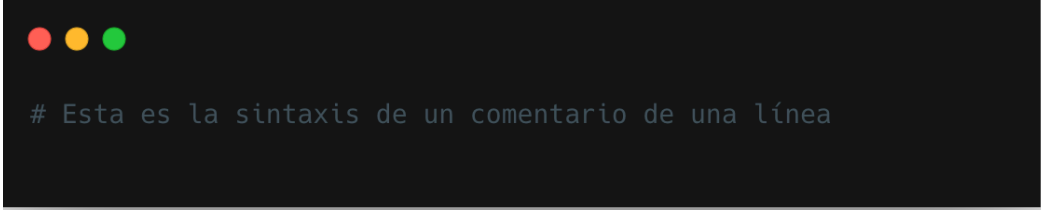
Los comentarios son una parte esencial del código en todos los lenguajes, no son otra cosa que *fragmentos de código que no se ejecutan.*

Son importantes ya que muchas veces se trabaja en equipo y es crucial aclarar qué tarea hace una función, por qué se nombró de tal o cual manera a una variable, titular un código modularizado, y también para dejar guardado código que funciona pero que en este momento ya no necesitamos, etc.

Hay tres tipos de comentarios: *de una línea, de media línea, de múltiples líneas.*

DE UNA LÍNEA


Aquellos comentarios que *ocupan una línea completa, y no comparten línea con código ejecutable.* Se utilizan para *dar un título o indicar algo del código de arriba o de abajo.*



```
# Esta es la sintaxis de un comentario de una línea
```

DE MEDIA LÍNEA

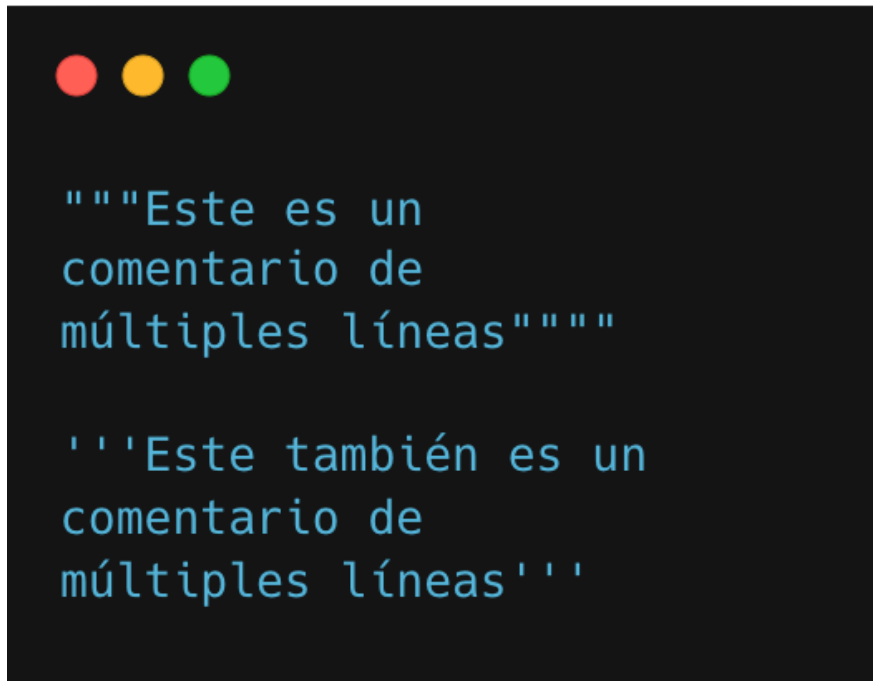
Aquellos comentarios que *ocupan parcialmente una línea y que comparten línea con fragmentos de código ejecutable*. Se utilizan para *indicar qué se hace en dicha línea de código y/o porqué*.



```
for i in range(10): # Este es la sintaxis de un comentario de media línea
```

DE MÚLTIPLES LÍNEAS

Aquellos comentarios que *ocupan más de una línea*. El entrecomillado puede ser simple o doble, siempre que se respeten el de apertura y el de cierre. Se utilizan para *indicar qué se hace en ese bloque de código y/o porqué* y también para *marcar el código que queremos dejar sin efecto*.




```
"""Este es un
comentario de
múltiples líneas"""

'''Este también es un
comentario de
múltiples líneas'''
```

DOCUMENTACIÓN

Además del código en general, hay algo muy importante que se denomina *documentación*. Son comentarios que indican qué se hace en cada sector del programa y por qué. Quizás ahora no lo vean como algo importante, pero muchas veces tendrán que revisar el código de otra persona y descifrar qué quiso hacer con tal o cual función, o de ustedes mismos y no recuerden. Si están dentro de una función e indicando qué es la acción o tarea que realiza ésta al ser llamada, dentro de una clase indicando a qué corresponde dicho blueprint o dentro de un bucle indicando qué es lo que se hará en cada iteración, se denominan *docstrings*.



```
def suma(a, b):  
    """Cada vez que la  
    función sea llamada  
    devolverá la suma entra  
    a y b.  
    :param a: sumando  
    :param b: sumando  
    :return: a + b"""  
  
    return a + b
```

Buenas prácticas

Este apartado es solo de sugerencias pero, si las aplican, verán que el código queda mucho más ordenado, limpio y legible. Para ello hay una guía de estilos denominada PEP 8 (Python Enhancement Proposal).

Todo eso lo pueden leer en la documentación oficial:

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

Unidad IV

DATOS, VARIABLES Y OPERADORES

Definiciones

DATO

Un dato, en programación, es la *expresión general que describe una o más características de la entidad sobre la que opera*; es decir, es la *mínima parte de la información*.

VARIABLE

Una variable es el *espacio de memoria que ocupa un dato al almacenarse*.

Posee un identificador (elegido por el programador) y un valor, y éstos se conectan con el operador de asignación =.

```
nombre_variable = valor
```

- *Nombre de la variable*: debe ser corto pero representativo, debe estar compuesto por letras (minúsculas y/o mayúsculas), números y underscores (_). Puede comenzar con una letra o un underscore, pero no con un número. Python es un lenguaje case sensitive, por lo tanto la variable hola es distinta de la variable Hola. Se deben evitar las keywords del lenguaje <https://docs.python.org/es/3/library/keyword.html>.

Ejemplos: todas los siguientes nombres de variables son *distintos*

```
var1
```

```
var_1
```

```
VAR1
```

```
VAR_1
```

`Var1`

`Var_1`

NOTA: en Python no existen las constantes, por lo tanto, según una convención de PEP8 toda variable que esté escrita de manera completa en MAYÚSCULAS se deberá tratar como constante.

- *Valor de la variable*: cualquier tipo que sea aceptado por Python, más adelante veremos cómo se representa cada uno. Es importante saber que el valor que se le asigna es el que luego se ejecutará, procesará y/o manipulará cada vez que se utilice o llame a la variable, salvo que éste cambie en algún momento.

Tipos de datos

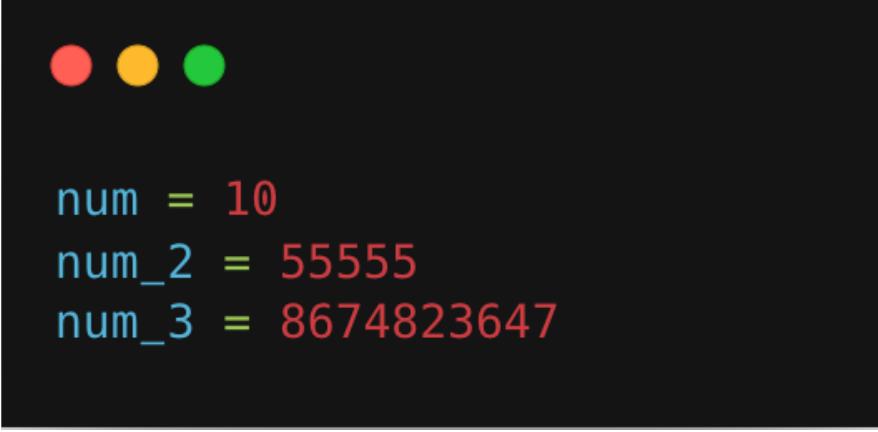
TIPOS DE DATOS PRIMITIVOS

Números

Todo dato con el que *se puedan realizar operaciones aritméticas*. Dentro de este tipo, hay tres subtipos:

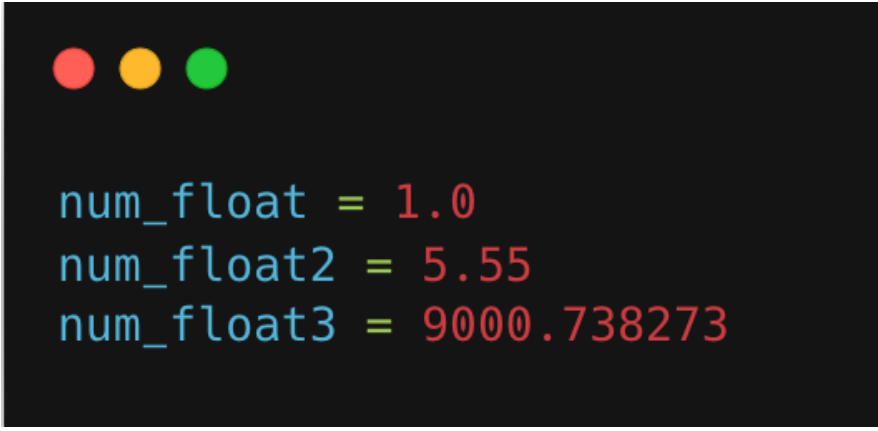
- `int`
- `float`

INT: todo número, ya sea positivo o negativo, que *no tiene parte flotante* (además del 0). En plataformas de 32 bits, int es todo número entero entre -2^{31} y $2^{31}-1$; y en plataformas de 64 bits, int es todo número entero entre -2^{63} y $2^{63}-1$.



```
num = 10
num_2 = 5555
num_3 = 8674823647
```

FLOAT: todo número, ya sea positivo o negativo, que tenga *al menos un número en la parte flotante, distinto o igual a 0*. Son aquellos que mal llamamos números decimales.




```
num_float = 1.0
num_float2 = 5.55
num_float3 = 9000.738273
```

Cadenas de texto (strings)


Las cadenas de texto, o cadenas de caracteres (dependiendo de la bibliografía) son *caracteres de cualquier tipo encerrados entre comillas, este entrecomillado indica que todos esos caracteres pertenecen al mismo dato*.

Las comillas a utilizar pueden ser dobles “” o simples “, no importa cuál se utilice siempre que se respete que el tipo que se use de apertura y de cierre coincida (PEP8 sugiere las simples). Esto hace que dentro de la cadena de caracteres se pueda también entrecomillar sin problema.



```
nombre = "Magali"  
apellido = 'Dominguez Lalli'
```

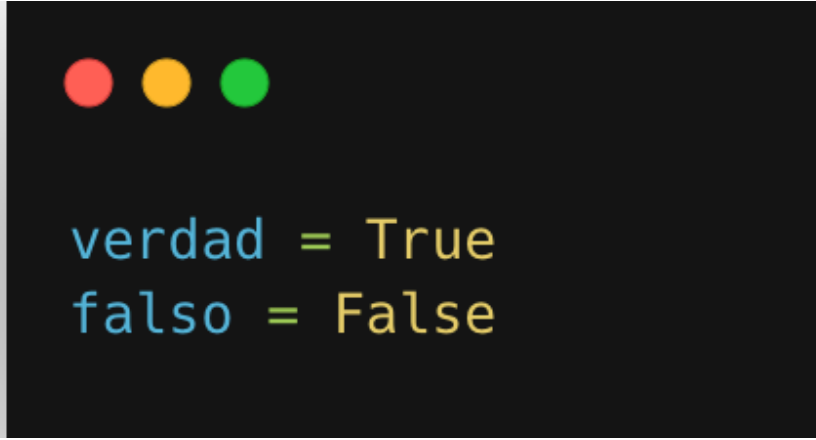
Dentro de las comillas se pueden agregar conjuntos de caracteres añadiendo adelante de éstos una backslash \. Como por ejemplo, para marcar un salto de línea se utiliza \n. Esto se denominan *secuencias de escape*.



```
info = "Mi nombre es Magalí\ny tengo 32 años"  
print(info)  
  
>>> Mi nombre es Magalí  
y tengo 32 años
```

Booleanos (bool)

Un tipo de dato booleano es aquel que *representa un valor de verdad*: verdadero (True) o falso (False). Es muy útil para trabajar con condicionales, bucles, clases y objetos, como veremos más adelante.




```
verdad = True
falso = False
```

COLECCIONES

Listas

La lista es un *tipo de colección ordenada mutable*, es decir, es una *colección de uno o más datos a los que se accede mediante un index (que comienza en 0) y puede ser modificada tantas veces como se desee*. Es similar a lo que en otros lenguajes se conoce como array, la diferencia es que una lista puede tener elementos de distinto tipo, mientras que un array debe respetar el tipo de elemento en todos los índices.



```
mi_lista = [1, "hola", True, 2.5]
```

constructor → **[]**

Para acceder a los elementos, debe indicarse el índice, siempre tener en cuenta que en Python, y en la mayoría de los lenguajes de programación, los índices comienzan en 0, no en 1.




```
mi_lista = [1, "hola", True, 2.5]

print(mi_lista[1])
print(mi_lista[-1])
print(len(mi_lista))

> hola
> 2.5
> 4
```

Se dice que son mutables porque se pueden modificar a lo largo de la ejecución del programa.



```
mi_lista = [1, "hola", True, 2.5]
mi_lista[1] = "hola mundo"
print(mi_lista)

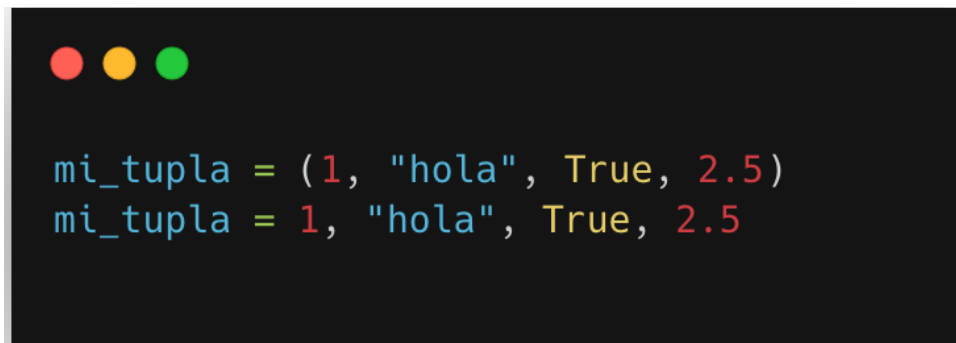
>>> [1, "hola mundo", True, 2.5]
```

Más adelante veremos los métodos para trabajar con datos de tipo *list*.

Tuplas

Una tupla es una colección ordenada inmutable, es decir, es una colección de uno o más datos a los que se accede mediante un index y no puede ser modificada una vez

que es creada.

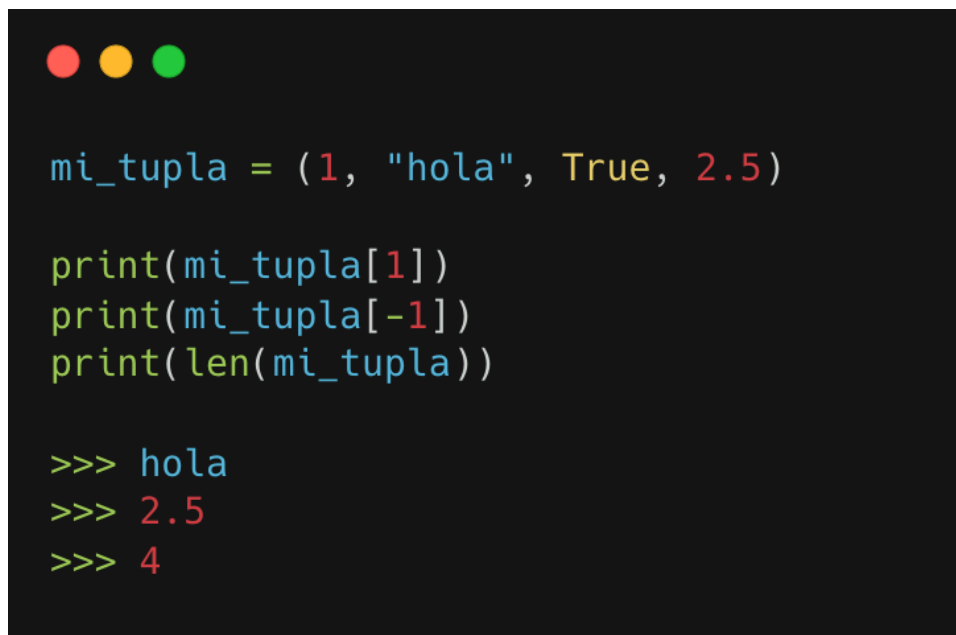


```
mi_tupla = (1, "hola", True, 2.5)
mi_tupla = 1, "hola", True, 2.5
```

constructor → ,

El constructor no es el paréntesis sino la coma, se utilizan los paréntesis porque cuando se ejecuta el programa el output es con paréntesis, entonces así es más ordenado.

Al igual que en las listas, para acceder a sus elementos, se debe indicar el índice.




```
mi_tupla = (1, "hola", True, 2.5)

print(mi_tupla[1])
print(mi_tupla[-1])
print(len(mi_tupla))

>>> hola
>>> 2.5
>>> 4
```

Son inmutables, por lo tanto, no se puede cambiar el valor de sus elementos.



```
mi_tupla = (1, "hola", True, 2.5)
mi_tupla[1] = "hola mundo"
print(mi_tupla)

>>> Traceback (most recent call last)
Cell In[2], line 2
      1 mi_tupla = (1, "hola", True,
2.5)
----> 2 mi_tupla[1] = "hola mundo"
      3 print(mi_tupla)

TypeError: 'tuple' object does not
support item assignment
```

Si lo dejamos como tupla y le damos run (ejecutar), nos saldrá un mensaje de error en la consola o en la terminal, según cuál estemos usando:

Es importante tener en cuenta que, si el uso que se le va a dar a una colección es muy básico y no presenta modificaciones, es preferible utilizar una tupla en vez de una lista ya que ocupan menos espacio de memoria.

Lo mismo que con las listas, más adelante veremos métodos para trabajar con ellas.

Diccionarios

Un diccionario, también llamado por varios autores *matriz asociativa*, es una *colección no ordenada y mutable de conjuntos clave-valor*. La clave se separa del valor mediante los dos puntos : y las claves son inmutables e irrepetibles pero los valores pueden ser tanto mutables como repetidos. La serie de elementos

clave-valor van entre corchetes. Su sintaxis es muy similar a los objetos JSON.

```
mis_datos = {  
    "nombre": "Magali",  
    "apellido": "Dominguez Lalli",  
    "edad": 32  
}
```

Al ser colecciones no ordenadas, no tienen índices, sino que se accede a sus elementos mediante los elementos clave:

```
mis_datos = {  
    "nombre": "Magali",  
    "apellido": "Dominguez Lalli",  
    "edad": 32  
}  
  
print(mis_datos["nombre"])  
  
>>> Magali
```

La otra condición es que sea mutable, por lo tanto se le pueden cambiar los valores a sus claves e incluso agregar o quitar pares clave-valor.

```
mis_datos = {  
    "nombre": "Magali",  
    "apellido": "Dominguez Lalli",  
    "edad": 32  
}  
  
mis_datos[edad] = 33  
print(mis_datos)  
  
>>> {"Magali", "Dominguez Lalli", 33}
```

Tal como con las listas y las tuplas, más adelante veremos métodos para trabajar con diccionarios.

Entrada y salida de datos

SALIDA (OUTPUT)

Como estuvimos viendo, para imprimir datos por consola, terminal y/o intérprete es necesario usar la función *print()*, que corresponde al grupo de built-in functions <https://docs.python.org/3/library/functions.html>.

Entonces, *la salida de datos se hace con print()*:

```
numero = 5

print(numero) # se imprime el dato almacenado en la variable numero
print(5) # se imprime un dato puro
print(len([1, 2, 3])) # se imprime el valor que devuelva la función len()
print(5 + 10) # se imprime el resultado de la expresión 5 + 10
```

ENTRADA (INPUT)

La entrada de datos se realiza con la built-in function `input()`, cuando aparece esta función, el programa le pide al usuario que ingrese el o los datos poniendo un cursor en el lugar donde debe/n ser ingresado/s.

```
input( )
```

```
>>>
```

También se le puede indicar al usuario qué es lo que se le solicita:

```
input("Ingrese su nombre: ")
```

```
>>> Ingrese su nombre:
```

IMPORTANTE: todo dato que se ingrese por teclado siempre será de tipo string, por más que sea un número, así que si se trabajará con un dato ingresado con `input()` y se desea que sea un número, antes de hacer cualquier operación, hay que convertirlo.

```
int(input("Ingrese un número: "))  
float(input("Ingrese un número: "))
```

Operadores

Un operador es un símbolo que se aplica a uno o varios operandos en una expresión o instrucción con el fin de obtener cierto resultado.

OPERADORES ARITMÉTICOS

Símbolos que se utilizan para realizar operaciones aritméticas y manipular datos de tipo `int` y `float`.

OPERADOR	TAREA	EJEMPLO	RESULTADO
+	suma	$a = 5 + 6$	11
-	resta	$b = 6 - 5$	1
-	negación	$c = -7$	-7
*	multiplicación	$d = 8 * 9$	72

**	potenciación	e = 2 ** 2	4
/	división	f = 6 / 3	2.0
//	división entera	g = 5 // 3	1
%	módulo	h = 7 % 2	1

Para hacer operaciones más avanzadas, se puede importar el módulo *math* y trabajar con sus funciones.

OPERADORES DE ASIGNACIÓN

Símbolos que se utilizan para asignar un valor a una variable.

OPERADOR	TAREA	EJEMPLO	RESULTADO
=	asignar a la variable de la izquierda el valor de la derecha	a = 5	5
+=	suma a la variable del lado izquierdo el valor del lado derecho	a += 5	10
-=	resta a la variable del lado izquierdo el valor del lado derecho	a -= 5	5
*=	multiplica a la variable del lado izquierdo por el valor del lado derecho	a *= 8	40
/=	divide a la variable del lado izquierdo por el valor del lado derecho	a /= 2	20.0
**=	eleva a la variable del lado izquierdo al exponente del valor del lado derecho	a **= 3	8000.0
//=	divide de manera entera a la variable del lado	g //= 4	2000

	izquierdo por el valor del lado derecho		
%=	devuelve el resto de la división entre la variable del lado izquierdo y el valor del lado derecho	h %= 2	0

OPERADORES RELACIONALES

Aquellos operadores que, como su nombre lo indica, *devuelven un valor booleano según la relación que haya entre los operandos*.

Todos darán como resultado True si efectivamente *cumplen su tarea*, de lo contrario darán como resultado False. Aquí vale la pena aclarar que los valores True y False rara vez los veremos asociados a una variable directamente sino que la mayoría de las veces serán el resultado de ciertas expresiones.

OPERADOR	TAREA	EJEMPLO
==	evalúa que los operandos sean iguales	a == b
!=	evalúa que los operandos sean distintos	b != c
<	evalúa que el operando de la izquierda sea menor que el operando de la derecha	d < e
>	evalúa que el operando de la izquierda sea mayor que el operando de la derecha	f > g
<=	evalúa que el operando de la izquierda sea menor o igual que el operando de la derecha	h <= i
>=	evalúa que el operando de la izquierda sea mayor o igual que el operando de la derecha	j >= k

OPERADORES LÓGICOS

Aquellos que *devuelven un resultado booleano si se cumple la función del operador*. Se aplican sobre valores booleanos.

OPERADOR	TAREA	EJEMPLO	RESULTADO
and	evalúa si todos los operandos involucrados son True	a and b	True sí y solo si todos los operandos involucrados tienen valor True
or	evalúa si al menos uno de los operandos involucrados es True	c or d	True sí y solo si al menos uno de los operandos involucrados tiene valor True
not	da lo opuesto al valor del operando involucrado	not e	True sí y solo si el operando involucrado tiene valor False

OPERADORES A NIVEL DE BIT

Las computadoras reciben información y la procesan en binario (unos y ceros), cada uno y cada cero que forme parte de un dato es un bit. Por ejemplo, el número binario 1000 (8 en sistema decimal), ocupa 4 bits y 1 byte. Un byte son ocho bits, y siempre se completa el octeto con ceros a la izquierda.

Por consiguiente, también hay operadores para trabajar con estos datos.

OPERADOR	TAREA	EJEMPLO
&	Compara los bits de cada operando y devuelve 1 cuando ambos sean 1, y 0 en otro caso, formando otro binario y convirtiéndolo a decimal	a & b
	Compara los bits de cada operando y devuelve 1 cuando alguno de los dos sea 1, y 0 si ambos son 0, formando otro binario y convirtiéndolo a	c d

	decimal	
~	cambia el valor de cada bit por su contrario, si es 1, lo cambia por 0 y si es 0 lo cambia por 1, creando otro binario y convirtiéndolo a decimal	~ e
<<	desplaza hacia la izquierda los bits que se indiquen, creando otro binario y convirtiéndolo a decimal	f << 2
>>	desplaza hacia la derecha los bits que se indiquen, creando otro binario y convirtiéndolo a decimal	g >> 2

Unidad V

ESTRUCTURAS DE CONTROL DE FLUJO

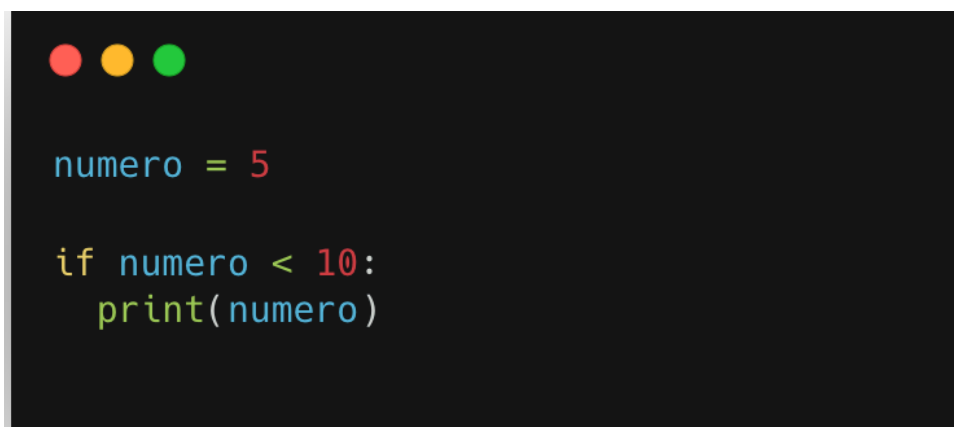
La mayoría de las veces que creamos un programa, necesitamos *chequear ciertas condiciones y ver qué camino seguir y/o cuántas veces*. Para esto, se utilizan las *estructuras de control de flujo*. Se llaman de esta manera porque lo que hacen es romper el flujo del programa.

CONDICIONALES

Los condicionales, como su nombre lo indica, *permiten comprobar condiciones y hacer que el programa se comporte de una forma u otra*, que ejecute un fragmento de código u otro, dependiendo del valor de verdad de esa condición.

Sentencia *if*

Es la *forma más simple de crear una estructura de flujo condicional*, *if*, en español “si”, seguido de la o las condiciones a evaluar, y luego dos puntos. En caso de que la o las condiciones resulte/n falsa/s, el programa seguirá a la próxima instrucción, dejando sin efecto el código dentro del *if*.

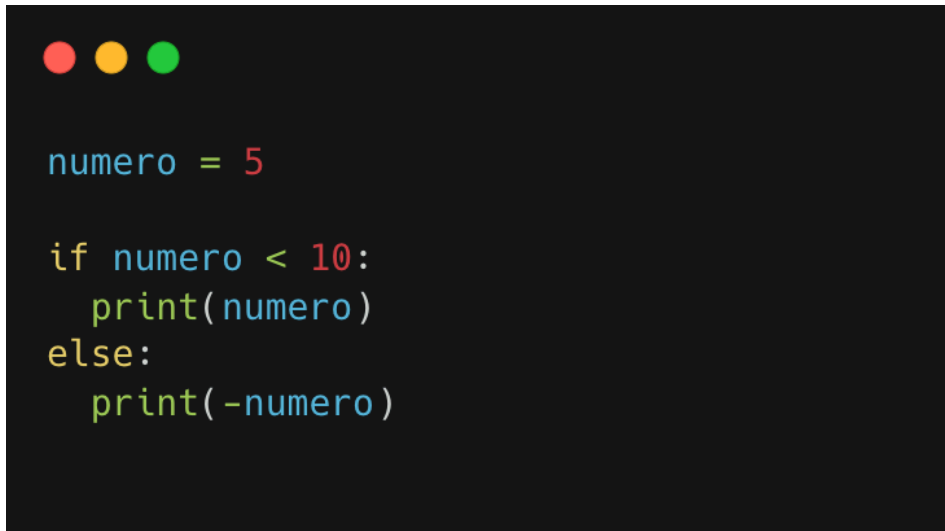
A screenshot of a code editor with a dark background. At the top left, there are three colored circles: red, yellow, and green. Below them, the following Python code is displayed:

```
numero = 5

if numero < 10:
    print(numero)
```

Sentencia *if-else*

Muchas veces, si la condición es falsa, no vamos a querer que el programa continúe a la siguiente instrucción sino que ejecute un código particular. Para eso existe la sentencia if-else, en español si-si no. *Si la condición es verdadera se ejecuta el código luego del if, si es falsa, se ejecuta el código luego del else.*

A screenshot of a terminal window with a dark background. At the top left, there are three colored circles: red, yellow, and green. The code is written in a monospaced font with syntax highlighting: 'numero' is blue, '=' is white, '5' is red, 'if' is blue, 'numero' is blue, '<' is white, '10' is red, ':' is white, 'print' is green, 'numero' is blue, 'else:' is yellow, and 'print' is green. The code is as follows:


```
numero = 5

if numero < 10:
    print(numero)
else:
    print(-numero)
```

Sentencia if-elif

En otros lenguajes elif sería else if, es decir, *es para crear un else (si la condición es falsa) y agregarle una condición a ese else*. Luego puede ir un else al final o no.

Hay que tener cuidado con el orden de las condiciones porque no necesariamente son mutuamente excluyentes. Es como un menú de opciones.



```
numero = 13


if numero < 10:
    print(numero)
elif 10 <= numero < 15:
    print(numero * 2)
elif numero == 20:
    print("veinte")
else:
    None
```

ITERATIVAS

Mientras que las estructuras de control de flujo condicionales permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, *las estructuras de control de flujo iterativas permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.* Estas estructuras son a menudo llamadas bucles.

Bucle while

El bucle *while*, en español *mientras*, ejecuta un fragmento de código mientras cierta condición permanece verdadera. Tener en cuenta que dentro de su estructura debe haber una instrucción que permita que la condición de verdad en algún momento se transforme en falsa.




```
numero = 10

while numero < 20:
    print(numero)
    numero += 1
```

Bucle *for*

A diferencia de otros lenguajes de programación, en Python *for* se utiliza para *iterar secuencias*. Ya sea para recorrer un iterable o bien para ejecutar un mismo fragmento de código una determinada cantidad de veces.



```
nombre = "Magali"

for letra in nombre:
    print(letra)
```




```
for i in range(10):  
    print("hola mundo")
```

PASS, BREAK & CONTINUE

Son sentencias que se usan mucho en las estructuras de control de flujo y también en la declaración de funciones y clases, y cada una tiene una función particular.

pass

Devuelve NULL al leer el interior de un bucle, como si estuviera vacío (que puede estarlo). Se suele utilizar cuando se está creando una función pero aún no se determinó su tarea, o una clase y aún no tiene atributos y métodos propios.

break

Termina el bucle actual y salta a la siguiente instrucción del programa.

continue

Salta a la siguiente iteración del bucle, ignorando la actual.

Unidad VI


FUNCIONES, MÉTODOS, FUNCIONES PREDEFINIDAS Y COMPRESIÓN DE LISTAS

Funciones

Una función es una *parte de un programa (subrutina) con un identificador, que puede ser invocada (llamada a ejecución) desde otras partes tantas veces como se desee, es decir, un bloque de código que puede ser ejecutado como una unidad funcional; opcionalmente puede recibir valores, se ejecuta y puede devolver un valor*. Desde el punto de vista de la organización, podemos decir que una función es algo que permite un cierto orden en el programa.

Utilidad

- Reutilizar el código, evitando repeticiones innecesarias.
- Ordenar el código.
- Modularizar: dividir el código en sectores.
- Reducir la probabilidad de errores




```
def suma(a, b):  
    """Cada vez que la  
    función sea llamada  
    devolverá la suma entra  
    a y b.  
    :param a: sumando  
    :param b: sumando  
    :return: a + b"""  
  
    return a + b
```

Los parámetros y el valor de retorno son opcionales. def y return son palabras reservadas de Python (palabras que no pueden ser reemplazadas por un valor determinado ni ser utilizadas para otra cosa que no sea para su función específica, para saber cuáles son todas se puede ejecutar el comando `help("keywords")` y el intérprete o terminal brindará la lista completa). El identificador de la función debe cumplir los mismos requisitos que los de las variables que ya vimos anteriormente.

Entonces:


- Parámetros: no son obligatorios, pero si la definición de función los tiene, la llamada debe tenerlos sí o sí.

Se pueden definir parámetros por default y, si la llamada no posee esos parámetros, se utilizan los de la declaración.



```
def suma(a=0, b=1):  
    sumatoria = a + b  
    return sumatoria
```

¿Qué pasa si no sé cuántos parámetros voy a necesitar? Fácil, *se indica que el número de argumentos es indefinido con un *args*. Importante: args es una convención por la abreviatura de argumento, pero puede ser el nombre de variable que deseen.



```
def suma(*args):  
    sumatoria = sum(args)  
  
    return sumatoria
```

- Valor de retorno: no es obligatorio pero, si está presente, para poder imprimirlo y visualizarlo es necesario no solo llamar a la función, sino también imprimir esa llamada; otra forma es almacenar la llamada en una variable e imprimir esa variable.


ÁMBITOS Y ALCANCES

En Python, como en la mayoría de los lenguajes, *las variables que se utilizan pertenecen a un ámbito en particular, por lo que se dice que tienen alcance local o alcance global (scopes).*

Esto es importante porque, dependiendo del alcance, se puede llamar o utilizar desde varios sectores del programa o no.

El intérprete de Python cuenta con un espacio de nombres en el que se ligan los objetos mediante la asignación de un nombre. Del mismo modo, las funciones crean su propio espacio de nombres, el cual deja de existir tan pronto como la función invocada concluye su ejecución.

A estos espacios de nombres diferenciados se les conoce como *ámbitos* y evita que objetos definidos con nombres idénticos dentro de una función sobrescriban el espacio de nombres del intérprete.

A screenshot of a Python interpreter window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in a syntax-highlighted font. It defines a global variable 'numero' with the value 10, then defines a function 'locales()' that sets a local variable 'numero' to 100 and prints it. After the function definition, the global 'numero' is printed, the function is called, and the global 'numero' is printed again. The interactive prompt shows the results: 10, 100, and 10.

```
numero = 10

def locales():
    numero = 100
    print(numero)

print(numero)
locales()
print(numero)

>>> 10
>>> 100
>>> 10
```

Con este ejemplo, pueden ver que la variable inicial es global y la variable dentro de la función es local, pertenece al ámbito de la función y una vez que ésta se termina de ejecutar queda sin efecto y no modifica a la global homónima.

PARÁMETROS Y ARGUMENTOS

Hay mucha bibliografía en inglés que toma como sinónimos parámetro y argumento. Pero me gustaría hacer una distinción para que la tengan en cuenta y les sea más fácil también entender la dinámica de una función.

- *Parámetros*: los datos de entrada que necesita una función para realizar su tarea. Son generales y están presentes en la definición de la función.

```
def suma(a, b):  
    return a + b
```

- *Argumentos*: los datos de entrada que efectivamente se le pasan a una función cuando es llamada. Son específicos y varían en cada llamada según se necesite.

```
def suma(a, b):  
    return a + b  
  
suma(10, 5)  
  
>>> 15
```

Métodos

Un método no es otra cosa que una función, la diferencia es que está definido dentro de cierta clase (una categorización con elementos que comparten características y comportamientos).

Cuando queremos averiguar un tipo de dato mediante la función `type()`, el intérprete nos devuelve `<class "tipodedato">`, es decir, cada tipo de dato es una clase, por lo tanto, si un método es una función perteneciente a cierta clase, es correcto hablar de métodos de strings, int, float, complex, etc.

Una función, como su nombre lo indica, cumple determinada tarea al ser ejecutada, un método, entonces, también.

Una forma muy útil de averiguar los métodos posibles es con la función

`help(tipodedato)`

MÉTODOS DE INT

Todo método aplicable a los números enteros.

A continuación, una lista de las más usadas:

MÉTODO	TAREA
<code>int()</code>	convierte a entero el dato que se pase por parámetro
<code>abs()</code>	devuelve el valor absoluto del dato que se pase por parámetro
<code>divmod(a, b)</code>	devuelve el cociente y resto de la división entre a y b
<code>float()</code>	convierte a float el dato que se pase por parámetro
<code>mod(a, b)</code>	devuelve el resto de la división entre a y b
<code>__neg__()</code>	devuelve el opuesto al dato que se pase por parámetro

pow(a, b)	devuelve el resultado de elevar a a la potencia b
round()	redondea el dato que se pase por parámetro
str()	convierte a string el dato pasado por parámetro
bit_lenght()	devuelve la cantidad de bits necesarios para convertir el número a binario
sum()	devuelve el resultado de la suma del conjunto que se pase por parámetro
bool()	devuelve True solo si el número es distinto de 0

MÉTODOS DE STRINGS

Todo método aplicable a datos de tipo string.

A continuación, una lista de las más usadas:

MÉTODO	TAREA
str()	convierte a string el dato que se pase por parámetro
__getitem__(a)	devuelve el caracter ubicado en el índice que se pasa por parámetro
len()	devuelve la cantidad de caracteres que tiene la cadena, contando espacios en blanco
capitalize()	devuelve el mismo string pero con la primera letra mayúscula
__contains__("a")	devuelve True si el caracter pasado por parámetro se encuentra en el string
isalnum()	devuelve True si la cadena es alfanumérica

isalpha()	devuelve True si la cadena es solo de caracteres del alfabeto
lower()	convierte a minúscula todos los caracteres de la cadena
upper()	convierte a mayúscula todos los caracteres de la cadena
title()	convierte a modo título todas las palabras contenidas en la cadena
istitle()	devuelve True si la cadena está en modo título
join(*args)	devuelve un nuevo string concatenado
strip()	devuelve el string sin los espacios en blanco al inicio y al final de la cadena
replace(a, b)	devuelve otro string con a reemplazado por b
rfind(a)	devuelve el último índice donde se encuentra a en el string
count(a)	devuelve el primer índice donde se encuentra a en el string
split()	devuelve una lista con cada caracter de la cadena como elemento
splitlines()	devuelve una lista con todas las líneas del string, el separador es el salto de línea

Acá quiero aclarar que para escribir strings con variables incluidas usaré la siguiente forma, que me parece la más práctica:

```
nombre = "Magali"
edad = 32

print(f"Mi nombre es {nombre} y tengo {edad}")

>>> Mi nombre es Magali y tengo 32
```

MÉTODOS DE LISTAS

Todo método aplicable a las listas.

A continuación, una lista de las más usadas:

MÉTODO	TAREA
list()	convierte el dato que se pase por parámetro en una lista, siempre que el dato sea iterable
__add__([lista2])	concatena la lista con la lista2, creando una nueva lista
__contains__(a)	devuelve True si la lista contiene al dato que se pasa por parámetro
__delitem__(i)	elimina el elemento que está en el índice que se pasa por parámetro
__getitem__(i)	devuelve el elemento unicado en el índice que se pasa por parámetro
len()	devuelve la cantidad de elementos que tiene la lista
__setitem__(i, a)	modifica el índice i por el elemento a
sizeof()	devuelve el espacio que ocupa la lista en la memoria

append(a)	agrega el elemento a a la lista, colocándolo al final
clear()	remueve todos los elementos de la lista
copy()	crea una copia de la lista, es útil cuando se quiere modificar algo pero dejar igual lo original
count(a)	devuelve la cantidad de veces que aparece el elemento a en la lista
pop()	elimina el último elemento de la lista
sort()	ordena en forma creciente los elementos de la lista, siempre que sean del mismo tipo
__reversed__()	crea una lista con los mismos elementos pero ordenados de forma inversa

MÉTODOS DE DICCIONARIOS

Todo método aplicable a los diccionarios.

A continuación, una lista de las más usadas:

MÉTODO	TAREA
dict()	convierte el dato que se pase por parámetro en un diccionario, siempre que se pueda
__delitem__(a)	elimina el par clave-valor correspondiente a la clave que se pasa por parámetro
__contains__(a)	devuelve True si el diccionario contiene como clave al dato que se pasa por parámetro
__getitem__(a)	devuelve el valor de la clave que se pasa por parámetro

len()	devuelve la cantidad de pares clave-valor que contiene el diccionario
__setitem__(a, b)	agrega el par clave-valor a-b al diccionario
sizeof()	devuelve el espacio de memoria que ocupa el diccionario
items()	devuelve una lista con tuplas de los pares clave-valor del diccionario
keys()	devuelve una lista con las claves del diccionario
clear()	remueve todos los pares clave-valor del diccionario
copy()	crea una copia de diccionario, es útil cuando se quiere modificar algo pero dejar igual lo original
values()	devuelve una lista con las claves del diccionario

Para ver todos los métodos de todos los tipos de datos, pueden visitar la sección documentation de la web oficial de Python:

<https://docs.python.org/3/tutorial/datastructures.html>.

Funciones predefinidas (built-in functions)

Las *funciones predefinidas*, en inglés *built-in functions*, son aquellas que ya vienen incluidas en Python. Por ejemplo, los métodos vistos anteriormente, no son otra cosa que funciones predefinidas.

Que ya estén incluidas en Python quiere decir que no se necesita incluir ningún módulo, paquete o librería para poder utilizarlas.

A continuación algunos ejemplos de las más usadas, además de los métodos que

ya vimos:

- `all()` → devuelve True si todos los elementos del objeto iterable que se le pasa por parámetro tienen un valor booleano True, False si tienen al menos un valor booleano False, 0 o vacío (que representa False).
- `any()` → devuelve True si al menos uno de los elementos del objeto iterable que se le pasa por parámetro tiene un valor booleano True.
- `ascii()` → devuelve el valor ASCII que corresponde al dato que se le pasa por parámetro.
- `bin()` → convierte en binario el entero que se le pasa por parámetro.
- `bool()` → convierte en booleano el elemento que se le pasa por parámetro.
- `bytearray()` → devuelve un array del número de bytes que se pasaron por parámetro.
- `byte()` → devuelve un objeto byte inmutable del elemento que se le pasó por parámetro.
- `callable()` → devuelve True si el objeto pasado por parámetro es “llamable”, las funciones son llamables por ejemplo.
- `chr()` → devuelve el caracter que corresponde al valor ASCII pasado por parámetro.
- `compile()` → convierte un string en un code object.
- `delattr()` → toma dos parámetros, una clase y un atributo de ella y elimina el atributo.
- `dir()` → devuelve todos los atributos del objeto que se le pase por parámetro.
- `enumerate()` → agrega un contador al objeto iterable.
- `eval()` → toma un string como parámetro y lo convierte en una expresión.
- `filter()` → se le pasan dos parámetros, una condición y un elemento iterable y devuelve aquellos en los cuales se cumple la condición.
- `format()` → es otra forma de escribir un string con variables, las cuales se pasan en orden.
- `frozenset()` → devuelve un conjunto inmutable del elemento que se le pasa por parámetro.
- `getattr()` → se le pasan dos parámetros, una clase y un atributo, y devuelve el valor de ese atributo.
- `hasattr()` → se le pasan dos parámetros, una clase y un atributo, y devuelve

True si dicha clase tiene ese atributo.


- `hash()` → devuelve el valor hash del dato que se pasa por parámetro.
- `hex()` → convierte a hexadecimal el dato que se le pasa por parámetro.
- `id()` → devuelve el valor identidad del dato que se le pasa por parámetro.
- `isinstance()` → se pasan dos parámetros, un dato y un tipo de objeto y devuelve True si ese dato pertenece a ese objeto.
- `issubclass()` → se pasan dos parámetros (dos clases) y devuelve True si la primera es subclase de la segunda.
- `iter()` → devuelve un iterador Python.
- `map()` → se le pasan dos parámetros, una función y un elemento iterable, y devuelve True o False en cada iteración si se cumple o no la función respectivamente.
- `max()` → devuelve el valor máximo de un conjunto.
- `min()` → devuelve el valor mínimo de un conjunto.
- `next()` → devuelve el próximo elemento del objeto iterable que se pasa por parámetro.
- `oct()` → convierte a octal el dato que se le pasa por parámetro.
- `open()` → abre el archivo de la ruta de acceso que se pasa por parámetro.
- `type()` → devuelve el tipo al que pertenece el dato pasado por parámetro.
- `ord()` → devuelve el valor Unicode al que pertenece el dato que se le pasó por parámetro.
- `range()` → crea una lista de elementos consecutivos, si se le pasa un sólo n parámetro es de 0 a n-1, si se le pasan dos parámetros n1 y n2, el rango es de n1 a n2-1, si se le pasan 3 parámetros n1, n2 y n3, el rango es de n1 a n2-1 y el salto es de n3 en n3.
- `set()` → devuelve un conjunto del dato que se le pasó por parámetro.
- `slice()` → devuelve los elementos del dato que forman parte de esa porción de dato, puede tener 2 o 3 parámetros, al igual que range.
- `sum()` → suma todos los valores de un iterable y devuelve el resultado.
- `zip()` → devuelve un objeto iterable de tuplas.

Compresión de listas

Hay diferentes formas de crear listas, pero, para entenderlas mejor existe la forma de *list comprehension*, es una suerte de combinación de listas con funciones y/o bucles.

Con este tipo de sintaxis se puede trabajar con cualquier tipo de dato, e incluso con archivos. Se utilizan mucho para filtrar data de archivos en data science.

Ejemplo: crear un código que nos muestre las potencias de los números del 0 al 9.



```
potencias = [pow(i, 2) for i in range(10)]  
  
print(potencias)  
  
>>> [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Unidad VII

CLASES, OBJETOS Y POO

Como ya vimos, Python es un lenguaje multiparadigma, sin embargo, *todo en Python es un objeto*.

CLASES Y OBJETOS

Una clase es un modelo o prototipo que define los atributos y métodos comunes a todos los objetos de cierto grupo, un blueprint.

En los lenguajes de programación convencionales los atributos corresponden a los valores que puede almacenar un objeto, mientras que los métodos son bloques de código que se ejecutan cuando son invocados. En Python ocurre algo muy similar, con la diferencia de que los atributos también son objetos y no sólo valores.

- *Atributo: características que tendrán los objetos pertenecientes a la clase.*
- *Métodos: comportamientos que tendrán los objetos pertenecientes a la clase.*

Un objeto es una entidad que agrupa un estado y una funcionalidad relacionadas. El estado del objeto se define a través de los atributos, mientras que la funcionalidad se modela a través de los métodos.

Cuando se crea un objeto, si dice que se instancia la clase a la que pertenece, y luego, para acceder a sus atributos y métodos se utiliza la nomenclatura del punto.

Vamos con un ejemplo fácil:


```

class Persona:

    def __init__(self, nombre, anio, dni):
        self.nombre = nombre
        self.anio = anio
        self.dni = dni

    def descripcion(self):
        print(f"{self.nombre} nació en {self.anio} y su dni es {self.dni}")

magali = Persona("Magali", 1991, 36382114)
magali.descripcion()

>>> Magali nació en 1991 y su dni es 36382114

```

En ambos pueden observar tres cosas:

- El constructor `__init__` : se ejecuta justo después de crear un nuevo objeto a partir de la clase, proceso que se conoce con el nombre de instanciación. El método `__init__` sirve, como sugiere su nombre, para realizar cualquier proceso de inicialización que sea necesario.
- Parámetro `self`: el primer parámetro de `__init__` y del resto de métodos de la clase es siempre `self`, y sirve para referirse al objeto actual. Este mecanismo es necesario para poder acceder a los atributos y métodos del objeto diferenciando, por ejemplo, una variable local `mi_var` de un atributo del objeto `self.mi_var`.
- Para crear un objeto se escribe el nombre de la clase seguido de cualquier parámetro que sea necesario entre paréntesis, excepto `self`. Estos parámetros son los que se crearán en el método `__init__`.

Herencia

En un lenguaje orientado a objetos, cuando hacemos que una clase (subclase) herede de otra clase (superclase) estamos haciendo que la subclase contenga todos los atributos y métodos que tenía la superclase. Esta acción también se suele llamar a menudo “extender una clase”.

¿Qué ocurriría si quisiéramos especificar un nuevo parámetro a la hora de crear una subclase? Bastaría con escribir un nuevo método `__init__` para ella que se ejecutaría en lugar del `__init__` de la superclase. Esto es lo que se conoce como sobrescribir métodos. Ahora bien, puede ocurrir en algunos casos que necesitemos sobrescribir un método de la clase madre, pero que en ese método queramos ejecutar el método de la clase madre porque nuestro nuevo método no necesite más que ejecutar un par de nuevas instrucciones extra. En ese caso usaríamos la sintaxis `super().__init__(self, args)` para llamar al método de igual nombre de la clase padre.

Veamos un ejemplo completo:

```
class Persona:

    def __init__(self, nombre, anio, dni):
        self.nombre = nombre
        self.anio = anio
        self.dni = dni

    def descripcion(self):
        print(f"{self.nombre} nació en {self.anio} y su dni es {self.dni}")

class Empleado(Persona):

    def __init__(self, nombre, anio, dni, profesion, puesto):
        super().__init__(nombre, anio, dni)
        self.profesion = profesion
        self.puesto = puesto

    def cargo(self):
        print(f"{self.nombre} es {self.profesion} y ocupa el puesto de {self.puesto}")

magali = Persona("Magali", 1991, 36382114)
juan = Empleado("Juan", 1980, 30344567, "contador", "gerente")
magali.descripcion()
juan.cargo()

>>> Magali nació en 1991 y su dni es 36382114
>>> Juan es contador y ocupa el puesto de gerente
```

Encapsulación

La encapsulación se refiere a impedir el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase.

En Python no existen los modificadores de acceso, y lo que se suele hacer es que el acceso a una variable o función viene determinado por su nombre: *si el nombre comienza con dos guiones bajos (y no termina también con dos guiones bajos) se trata de una variable o función privada, en caso contrario es pública.*

```
class Persona:

    def __init__(self, nombre, anio, dni):
        self.nombre = nombre
        self.anio = anio
        self.__dni = dni

    def descripcion(self):
        print(f"{self.nombre} nació en {self.anio} y su dni es {self.__dni}")

    def doc(self):
        return self.__dni
```

En este ejemplo dni es un atributo encapsulado, por lo tanto solo puedo acceder desde la clase y no desde fuera. Sin embargo, podemos crear un método cuyo único fin sea devolver la información de dicho atributo, cosa que es muy común.

Polimorfismo

El polimorfismo se refiere a la habilidad de objetos de distintas clases de responder al mismo mensaje. Esto se puede conseguir a través de la herencia: un objeto de una clase derivada es al mismo tiempo un objeto de la clase madre, de forma que allí donde se requiere un objeto de la clase madre también se puede utilizar uno de la clase hija.

.

```
class Persona:

    def __init__(self, nombre, anio, dni):
        self.nombre = nombre
        self.anio = anio
        self.dni = dni

    def descripcion(self):
        print(f"{self.nombre} nació en {self.anio} y su dni es {self.dni}")

class Empleado(Persona):

    def __init__(self, nombre, anio, dni, profesion, puesto):
        super().__init__(nombre, anio, dni)
        self.profesion = profesion
        self.puesto = puesto

    def descripcion(self):
        print(f"{self.nombre} es {self.profesion} y ocupa el puesto de {self.puesto}")
```

En este caso, el método `descripcion()` para cualquier objeto de la clase `Empleado` será el definido en ella. Siempre el programa lo que hará es buscar el método en la clase en la que está definida el objeto y, en caso de no encontrarlo, recién ahí buscarlo en la clase madre.

```
class Perro:

    def __init__(self, nombre):
        self.nombre = nombre

    def desplazarse(self):
        print(f"{self.nombre} se mueve en 4 patas")

class Pez:

    def __init__(self, nombre):
        self.nombre = nombre

    def desplazarse(self):
        print(f"{self.nombre} se mueve nadando")

canela = Perro("Canela")
nemo = Pez("Nemo")
canela.desplazarse()
nemo.desplazarse()

>>> Canela se mueve en 4 patas
>>> Nemo se mueve nadando
```

Cuando son clases independientes, como en el ejemplo anterior, el método que se ejecutará es el definido en la clase en la que esté instanciada el objeto. Es decir, poco tiene que ver con el identificador del método sino con la clase a la que éste pertenece.

PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

Como hemos podido constatar, la programación orientada a objetos es un paradigma de programación que busca representar entidades u objetos agrupando datos y métodos que puedan describir sus características y comportamientos.

En este paradigma, los conceptos del mundo real relevantes para nuestro problema a resolver se modelan a través de clases y objetos, y el programa

consistirá en una serie de interacciones entre dichos objetos.

MÉTODOS ESPECIALES

Así como vimos el método constructor `__init__` hay otros que, por supuesto, cumplen otras funciones, a continuación los que son, quizás, lo más utilizados.

- `__new__(cls, args)`: método exclusivo de las clases que se ejecuta antes que `__init__` y que se encarga de construir y devolver el objeto en sí. Se trata de un método estático, es decir, que existe con independencia de las instancias de la clase: es un método de clase, no de objeto, y por lo tanto el primer parámetro no es `self`, sino la propia clase: `cls`.
- `__del__(self)`: método llamado cuando el objeto va a ser borrado. También llamado destructor, se utiliza para realizar tareas de limpieza.
- `__str__(self)`: método llamado para crear una cadena de texto que represente a nuestro objeto. Se utiliza cuando usamos `print` para mostrar nuestro objeto o cuando usamos la función `str(obj)` para crear una cadena a partir de nuestro objeto. Le podemos dar el formato que deseemos.
- `__cmp__(self, otro)`: método llamado cuando se utilizan los operadores de comparación para comprobar si nuestro objeto es menor, mayor o igual al objeto pasado como parámetro. Debe devolver un número negativo si nuestro objeto es menor, cero si son iguales, y un número positivo si nuestro objeto es mayor. Si este método no está definido y se intenta comparar el objeto mediante los operadores `<`, `<=`, `>` o `>=` se lanzará una excepción. Si se utilizan los operadores `==` o `!=` para comprobar si dos objetos son iguales, se comprueba si son el mismo objeto (si tienen el mismo id).
- `__len__(self)`: método llamado para comprobar la longitud del objeto. Se utiliza, por ejemplo, cuando se llama a la función `len(obj)` sobre nuestro objeto. Como es de suponer, el método debe devolver la longitud del objeto.

Existen bastantes más métodos especiales que permiten, entre otras cosas, utilizar el mecanismo de slicing sobre nuestro objeto, utilizar los operadores aritméticos o usar la sintaxis de diccionarios, los pueden encontrar en la web oficial de Python <https://docs.python.org/3/reference/datamodel.html>.

Unidad VIII

ARCHIVOS, MÓDULOS Y PAQUETES

Archivos

Un archivo es una secuencia de datos almacenados en un medio persistente que están disponibles para ser utilizados por un programa. Todos los archivos tienen un nombre y una ubicación dentro del sistema de archivos del sistema operativo.

Un programa no puede manipular los datos de un archivo directamente. Para hacerlo, debe abrir el archivo y asignarlo a una variable, que llamaremos el **archivo lógico**. Todas las operaciones sobre un archivo se realizan a través de esta variable.



Si el archivo no está previamente creado, se crea al abrirse, esto quiere decir que en el proceso creación y apertura pueden ser una sola etapa, siempre que la forma de apertura sea en alguno de los modos de escritura.

Cuando el archivo se crea y abre, se debe especificar para qué se abre: lectura, escritura, agregar contenido al final o lectura + escritura.

Modos de abrir un archivo

Por el tipo de archivo

- 't' se trata de un archivo de texto.
- 'b' permite escritura en modo binario
- 'U' define saltos de línea universales para el modo de lectura.

Por el tipo de acceso

- 'r' es el modo de solo lectura
- 'w' es un modo solo de escritura. En caso de existir un archivo.
- 'a' es un modo solo de escritura. En caso de existir un archivo, posiciona el cursor al final del contenido del mismo.
- 'r+' es un modo de lectura+escritura, el archivo debe existir.
- 'w+' es un modo de escritura+lectura, si el archivo no existe se crea al abrirse.

CREAR Y ABRIR ARCHIVO

A terminal window with a black background and three colored window control buttons (red, yellow, green) in the top left corner. It displays two lines of Python code: `with open('archivo.txt', 'w+')` on the first line and `pass` on the second line, both in a yellow monospace font.

```
with open('archivo.txt', 'w+'):
    pass
```

`with` es la sentencia que utilizaremos para poder trabajar con archivos. `open()` es el método que nos permite abrir un archivo, el primer parámetro es la ruta de acceso del archivo (si ésta es una ruta completa, es decir está fuera de la carpeta en la que estamos trabajando, se debe poner una `r` adelante `r'rutadeacceso'` para evitar que tome por ejemplo algún carácter como secuencia de escape); el segundo parámetro es el modo de abrir el archivo. Lo que va dentro de la sentencia `with` es la manipulación que se hace del archivo. Una vez que se sale de la indentación el archivo se cierra.

MÉTODOS PARA MANIPULAR ARCHIVOS

A continuación, los métodos más comunes para manipular archivos:

- `write()`: escribe desde la primera línea del archivo, si éste ya tiene contenido, lo que se pase como parámetro lo sobrescribe. Importante, si el archivo se abre en “a”, cuando se utiliza el método `write()` éste agrega contenido al final del archivo, y agrega ese contenido tantas veces como se le dé run al programa.
- `read()`: lee el contenido del archivo, pero no lo imprime, para poder imprimirlo hay que almacenarlo en una variable e imprimirla.
- `seek()`: ubica el cursor en el índice que se le pase por parámetro.
- `readlines()`: convierte el contenido del archivo en una lista, el separador de cada elemento es el salto de línea, entonces cada elemento es una línea del archivo.
- `readline()`: lee la línea que se le pasa por parámetro.
- `writeline()`: escribe el archivo línea a línea.
- `writable()`: devuelve True si el archivo está abierto en modo escritura.
- `readable()`: devuelve True si el archivo está abierto en modo lectura.
- `seekable()`: devuelve True si es posible desplazarse dentro del archivo.
- `tell()`: devuelve la posición en la que se encuentra el puntero dentro del archivo.
- `close()`: cierra el archivo.

Aquellos que están en amarillo son para todo tipo de archivos y los que están en violeta son para archivos de texto.

Para más información: <https://docs.python.org/3/library/filesys.html>.

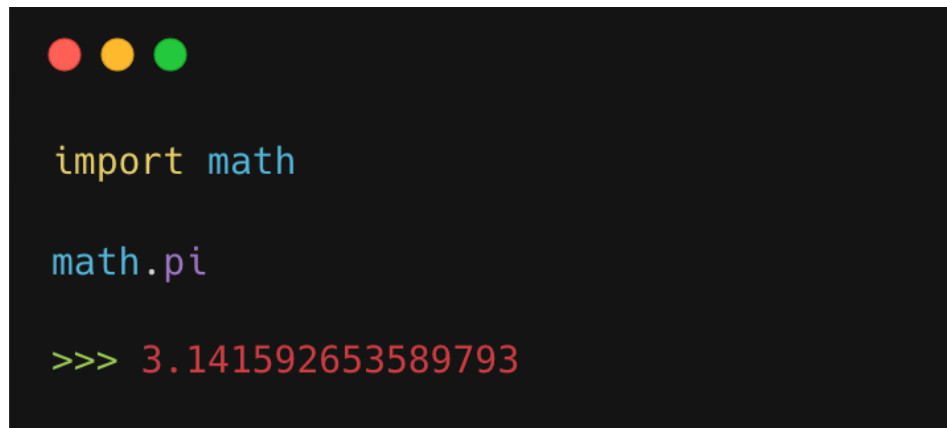
Módulos y paquetes

MÓDULOS

Un módulo es un tipo de archivo .py o .pyc que contiene funciones, clases, objetos, variables e incluso otros módulos para ser utilizados en otros archivos de Python.

Se utiliza para que el código sea más ordenado, limpio, legible y reutilizable.

Para poder usar el contenido de un módulo es necesario importarlo en el archivo en el que haremos uso del mismo.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays the following text:

```
import math  
  
math.pi  
  
>>> 3.141592653589793
```

Hay varias formas de importación y de éstas depende cómo se llame al contenido de cada módulo:

1. `import <nombre_módulo>`

Es la más simple. Cuando la importación se realiza de esta manera, para poder utilizar el contenido que contiene el módulo se hace de la siguiente forma:

`nombre_módulo.nombre_contenido`

2. `import <nombre_módulo> as <alias>`

Cuando la importación se realiza de esta manera, para poder utilizar el contenido que contiene el módulo se hace de la siguiente forma:

`alias.nombre_contenido`

3. `from <nombre_módulo> import <nombre_contenido>`

Cuando sabemos que voy a utilizar solo un contenido en particular, o más de uno, utilizamos esta manera de importar. Para poder usar dicho contenido, simplemente se lo llama por su nombre:

```
nombre_contenido
```

```
4. from <nombre_módulo> import *
```

Es igual a 1. pero la diferencia es que así el contenido se llama solo por su nombre, sin necesidad de nombrar el módulo, como si estuviera explícito en el archivo en el que estamos trabajando.

```
nombre_contenido
```

```
nombre_contenido2
```

```
nombre_contenido3
```

PAQUETES

Un paquete es un directorio o carpeta donde se almacenan diferentes módulos que están relacionados entre sí.

Se crean con una carpeta que tenga sí o sí un archivo (file) llamado `__init__.py`.
Dentro de un paquete puede haber otros paquetes, es importante que cada paquete interno tenga su propio file `__init__.py`.

Si el paquete no está en nuestro S.O. debemos primero instalarlo utilizando *pip* `install <nombre_paquete>` en nuestra terminal:

```
(base) magalidominguezlalli@Magalis-MacBook-Air ~ % pip install tensorflow
Collecting tensorflow
  Downloading tensorflow-2.16.1-cp39-cp39-macosx_12_0_arm64.whl (227.0 MB)
    - 3.6/227.0 MB 585.9 kB/s eta 0:06:22
```

Para importar es igual a los módulos, la única diferencia es que hay que revisar su documentación oficial y ver cómo indica la misma que hay que importarlos. La mayoría tiene una forma específica, que puede ser, por ejemplo, a través de un alias. Es importante seguir esta rigurosidad ya que son convenciones aceptadas y conocidas por todos.

How to import NumPy

To access NumPy and its functions import it in your Python code like this:

```
import numpy as np
```

We shorten the imported name to `np` for better readability of code using NumPy. This is a widely adopted convention that you should follow so that anyone working with your code can easily understand it.

MÓDULO `os`

El módulo `os` provee de funciones para interactuar con el sistema operativo (operative system).

Descripción	Método
Saber si se puede acceder a un archivo o directorio	<code>os.access(path, modo_de_acceso)</code>
Conocer el directorio actual	<code>os.getcwd()</code>
Cambiar de directorio de trabajo	<code>os.chdir(nuevo_path)</code>
Cambiar al directorio de trabajo raíz	<code>os.chroot()</code>
Cambiar los permisos de un archivo o directorio	<code>os.chmod(path, permisos)</code>
Cambiar el propietario de un archivo o directorio	<code>os.chown(path, permisos)</code>

Crear un directorio	<code>os.mkdir(path[, modo])</code>
Crear directorios recursivamente	<code>os.makedirs(path[, modo])</code>
Eliminar un archivo	<code>os.remove(path)</code>
Eliminar un directorio	<code>os.rmdir(path)</code>
Eliminar directorios recursivamente	<code>os.removedirs(path)</code>
Renombrar un archivo	<code>os.rename(actual, nuevo)</code>
Crear un enlace simbólico	<code>os.symlink(path, nombre_destino)</code>

os.path

El módulo os también nos provee del submódulo path (os.path) el cual nos permite acceder a ciertas funcionalidades relacionadas con los nombres de las rutas de archivos y directorios.

Descripción	Método
Ruta absoluta	<code>os.path.abspath(path)</code>
Directorio base	<code>os.path.basename(path)</code>
Saber si un directorio existe	<code>os.path.exists(path)</code>
Conocer último acceso a un directorio	<code>os.path.getatime(path)</code>

Conocer tamaño del directorio	<code>os.path.getsize(path)</code>
Saber si una ruta es absoluta	<code>os.path.isabs(path)</code>
Saber si una ruta es un archivo	<code>os.path.isfile(path)</code>
Saber si una ruta es un directorio	<code>os.path.isdir(path)</code>
Saber si una ruta es un enlace simbólico	<code>os.path.islink(path)</code>
Saber si una ruta es un punto de montaje	<code>os.path.ismount(path)</code>

MÓDULO SYS

El módulo sys es el encargado de proveer variables y funcionalidades, directamente relacionadas con el intérprete.

Variable	Descripción
<code>sys.argv</code>	Retorna una lista con todos los argumentos pasados por línea de comandos. Al ejecutar <code>python modulo.py arg1 arg2</code> , retornará una lista: <code>['modulo.py', 'arg1', 'arg2']</code>
<code>sys.executable</code>	Retorna el path absoluto del binario ejecutable del intérprete de Python
<code>sys.maxint</code>	Retorna el número positivo entero mayor, soportado por Python
<code>sys.platform</code>	Retorna la plataforma sobre la cual se está ejecutando

	el intérprete
<code>sys.version</code>	Retorna el número de versión de Python con información adicional

Método	Descripción
<code>sys.exit()</code>	Forzar la salida del intérprete
<code>sys.getdefaultencoding()</code>	Retorna la codificación de caracteres por defecto
<code>sys.getfilesystemencoding()</code>	Retorna la codificación de caracteres que se utiliza para convertir los nombres de archivos unicode en nombres de archivos del sistema
<code>sys.getsizeof(object[, default])</code>	Retorna el tamaño del objeto pasado como parámetro. El segundo argumento (opcional) es retornado cuando el objeto no devuelve nada.


Unidad IX

EXCEPCIONES

Excepciones

Una excepción es un error que ocurre durante la ejecución de un programa pero que no necesariamente está en la sintaxis del código.

Para entenderlo, vamos con un ejemplo sencillo: la división por 0. En este caso, la sintaxis del código es correcta, pero hay algo que igual hace que el código se rompa.



```
def division(a, b):  
    return a / b  
  
division(5, 0)  
  
>>> Traceback (most recent call last)  
      1 def division(a, b):  
      2     return a / b  
----> 5 division(5, 0)  
  
ZeroDivisionError: division by zero
```


Este error puede ser contemplado en el código para que no ocurra y el programa siga su curso sin mayores problemas.

```
def division(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "No se puede dividir por 0"

division(5, 0)

>>> No se puede dividir por 0
```

La estructura dentro del `try` se ejecutará *SIEMPRE*, si no hay error se sigue a la próxima instrucción de programa, si se encuentra con el error `ZeroDivisionError` se ejecutará, entonces, su bloque de código.

Podríamos también no detallar de qué error se trata y, en este caso, solo nos limitamos a marcar el error.

```
def division(a, b):
    try:
        return a / b
    except:
        return "Hay un error"

division(5, 0)

>>> Hay un error
```

Podemos tener tantos except como errores querramos contemplar, y al final un except general que sirva para indicar que no se encontraron ninguno de los otros errores pero que sigue habiendo uno.



```
def funcion():
    try:
        pass
    except RuntimeError:
        pass
    except TypeError:
        pass
    except ValueError:
        pass
    except:
        pass
```

Por supuesto que la única excepción no es la división por 0, pero sí la más conocida.

Algunos ejemplos de las más comunes:

- **ValueError**: cuando se espera un tipo de dato y se recibe otro.
- **NameError**: cuando se recibe el nombre de una variable, función, clase u objeto pero no fue previamente definido.
- **TypeError**: cuando el tipo de dato que se quiere manipular debería ser otro.