

# **Reducción de datos CCD con IRAF y Python**

Alexis Andrés

26 de agosto de 2024

# Índice general

<b>1</b>	<b>Introducción a Python</b>	<b>3</b>
	<b>Clase 1   Introducción a Python</b>	<b>4</b>
1	Nociones básicas . . . . .	4
1.1	Python en diferentes sistemas operativos . . . . .	4
1.1.1	Python en Windows . . . . .	5
1.1.2	Python en Linux . . . . .	5
1.1.3	Python en Mac OSX . . . . .	6
1.2	Ejecutando Python . . . . .	7
1.2.1	Creando un entorno virtual . . . . .	8
1.2.2	El editor Geany . . . . .	9
1.2.3	Jupyter Notebook . . . . .	11
1.2.4	Visual Studio Code y PyCharm . . . . .	13
1.3	Python como calculadora . . . . .	13
1.3.1	Operaciones aritméticas . . . . .	13
1.3.2	Operaciones matemáticas avanzadas . . . . .	15
	<b>Clase 2   Variables y contenedores</b>	<b>17</b>
2	Introducción . . . . .	17
2.1	Variables . . . . .	17
2.1.1	Variables tipo int/float . . . . .	18
2.1.2	Variables de tipo string . . . . .	18
2.1.3	Comentarios . . . . .	19
2.1.4	Variables tipo bool . . . . .	20
2.2	Contenedores . . . . .	21
2.2.1	Listas . . . . .	21
2.2.2	Tuplas . . . . .	24
2.2.3	Conjuntos . . . . .	25

2.2.4	Diccionarios	26
<b>Clase 3   Control de flujo y lógica</b>		<b>28</b>
3	Controles de flujo	28
3.1	Condicionales	28
3.1.1	Declaraciones if	28
3.1.2	Declaraciones if-else	29
3.1.3	Declaraciones if-elif-else	30
3.1.4	Expresiones if-else	31
3.2	Bucles	32
3.2.1	Bucle while	32
3.2.2	Bucle for	34
3.2.3	Contenedores por comprensión	37
<b>Clase 4   Funciones</b>		<b>39</b>
4	Funciones en Python	39
4.1	Funciones predefinidas	39
4.2	Funciones definidas por el usuario	40
4.2.1	Funciones sin parámetros	41
4.2.2	Funciones con parámetros	42
4.2.3	Funciones que devuelven un valor	42
4.3	Más sobre funciones con parámetros	43
4.3.1	Argumentos con un valor predeterminado	44
4.3.2	Argumentos de palabras clave	45
4.4	Problemas	47

# **Unidad 1**

## **Introducción a Python**

# Clase 1 | Introducción a Python

## 1. Nociones básicas

*Python* es un lenguaje de programación interpretado de alto nivel cuyas características permiten su uso en diversas disciplinas. Los lenguajes de alto nivel en programación poseen una sintaxis que se asemeja al lenguaje humano, lo que facilita la claridad y legibilidad del código, haciendo su escritura y comprensión más accesibles. En contraste, los lenguajes de bajo nivel, como el *ensamblador*, utilizan una sintaxis más cercana al lenguaje máquina, lo que los hace más eficientes para la computadora pero más complejos de leer y escribir para los humanos.

Python, al ser un lenguaje interpretado, traduce y ejecuta el código línea por línea durante la ejecución, lo que puede resultar en tiempos de ejecución más largos en comparación con los lenguajes compilados, que se traducen en código máquina antes de ejecutarse. Sin embargo, esta diferencia en el tiempo de ejecución suele ser insignificante en muchos casos prácticos. Además, el tiempo requerido para desarrollar y mantener código en Python es considerablemente menor comparado con los lenguajes de bajo nivel, equilibrando así el tiempo total de desarrollo.

Adicionalmente, Python permite la integración de bibliotecas y códigos escritos en otros lenguajes de programación, aprovechando sus ventajas y capacidades para mejorar el rendimiento. Esta flexibilidad y facilidad de uso han contribuido a que Python se convierta en uno de los lenguajes de programación más populares en la actualidad, especialmente en los campos de ciencias e ingeniería.

En esta primera clase revisaremos algunos conceptos básicos de Python, cómo instalarlo en cualquier sistema operativo y además elegiremos un entorno de desarrollo integrado para escribir nuestros códigos Python.

### 1.1. Python en diferentes sistemas operativos

Python es un lenguaje de programación multiplataforma, esto significa que puede utilizarse en todos los sistemas operativos principales. Sin embargo, los métodos de instalación difieren

dependiendo de cada sistema operativo. El objetivo de esta sección es que puedas ejecutar el famoso programa «¡Hola Mundo!» usando Python en tu sistema operativo. Ten en cuenta que trabajaremos exclusivamente con Python 3.

### 1.1.1. Python en Windows

Por lo general, Windows no cuenta con una versión de Python por defecto, por lo que será necesario instalarlo además de un editor de texto. Para verificar si tu versión de Windows cuenta con Python, primero debes abrir el menú de inicio, escribir los caracteres `cmd` y hacer clic sobre la aplicación llamada *símbolo del sistema*. Esto abrirá una terminal de comandos en la que deberás escribir las palabras `python --version` (todo en minúsculas) y presiona la tecla Enter.

Si el resultado es algo similar a `3.x.x`, entonces Python ya está instalado en tu sistema. Sin embargo, si el resultado es un mensaje que dice que `python` no es un comando reconocido, entonces sigue las siguientes instrucciones para instalarlo.

1. **Descarga el instalador de Python.** Abre tu navegador y dirígete a la página <https://www.python.org/downloads/>. Haz clic en el botón que dice «Download Python 3.x.x» (donde «3.x.x» es la versión más reciente).
2. **Ejecuta el instalador.** Encuentra el archivo del instalador que acabas de descargar (normalmente estará en tu carpeta de descargas) y haz doble clic para ejecutarlo. En la primera ventana del instalador, asegúrate de marcar la casilla que dice «Add Python 3.x to PATH». Esto es importante para que puedas usar Python desde terminal de comandos sin problemas. Finalmente, haz clic en «Install Now» para comenzar la instalación.
3. **Verifica la instalación.** Una vez que finalice la instalación, abre una nueva terminal de comandos y escribe las palabras `python --version`. Esto debería mostrar la versión de Python que has instalado.

### 1.1.2. Python en Linux

Cualquier sistema operativo Linux cuenta con una versión de Python instalada por defecto. Por lo tanto, solo necesitas verificar cuál versión tienes instalada en tu sistema. Para lograrlo,

abre una terminal de comandos con la combinación de teclas Ctrl+Alt+T y escribe las palabras **python --version** (todo en minúsculas) y presiona la tecla Enter. Si el resultado es un mensaje similar a **2.x.x** o si en cambio te aparece un mensaje que dice que el comando python no fue encontrado, entonces intenta escribiendo **python3 --version** y presiona Enter. El resultado debería ser algo como **3.x.x**.

### 1.1.3. Python en Mac OSX

La mayoría de los sistemas Mac OSX cuenta con al menos una versión de Python instalada por defecto. Esto significa que es posible que cuentes con dos versiones, para verificarlo, abre una terminal de comandos haciendo clic en **Aplicaciones > Utilidades > Terminal**. Ahora escribe las palabras **python --version** (todo en minúsculas) y presiona la tecla Enter.

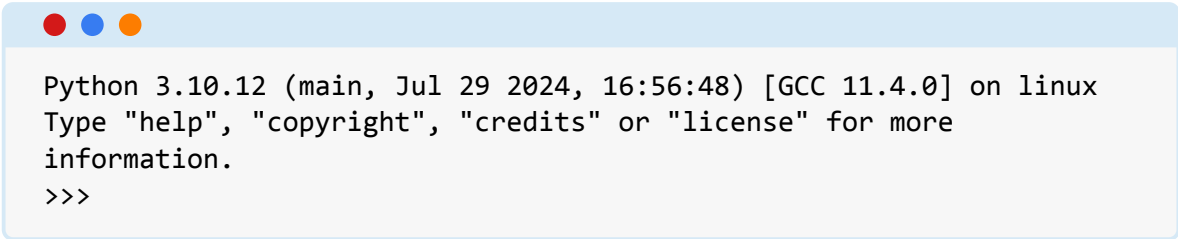
Al igual que con los sistemas Linux, es posible que el resultado sea un error o un mensaje con algo similar a **2.x.x**, lo que significa que la versión instalada es Python 2. Si cualquiera de las dos opciones anteriores es cierta, intenta escribiendo en la terminal **python3 --version** y presiona Enter. Si el resultado es algo similar a **3.x.x**, entonces cuentas con una versión de Python 3.

Si por algún motivo al escribir **python3 --version** obtienes un error, entonces necesitas instalar Python 3 en tu sistema. Para eso, sigue los siguientes pasos.

1. **Descarga el instalador.** Dirígete a la página <https://www.python.org/>. Coloca el cursor sobre el botón «Downloads» y selecciona «macOS». En la página de descargas deberías ver un botón llamado «**macOS 64-bit universal2 installer**» debajo del título «Stable Releases». Haz clic sobre ese botón para descargar el instalador con extensión **.pkg**
2. **Ejecuta el instalador.** Una vez que el archivo **pkg** se haya descargado, haz doble clic en él para iniciar el proceso de instalación. Aparecerá una ventana del asistente de instalación. Sigue las instrucciones en pantalla. Generalmente, solo necesitas hacer clic en «Continuar» y luego en «Instalar». El instalador pedirá tu contraseña de administrador para proceder. Una vez que se complete el proceso, haz clic en «Cerrar» para terminar.
3. **Verifica la instalación.** Abre una nueva terminal de comandos y escribe las palabras **python3 --version**. El resultado debería ser la versión de Python que acabas de instalar.

## 1.2. Ejecutando Python

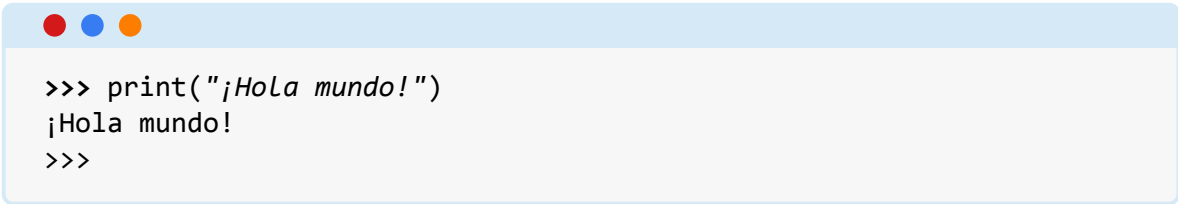
Para comenzar a usar Python, solo tienes que abrir una terminal de comandos, escribir la palabra **python3** y presionar la tecla Enter (Nota: este documento fue escrito en un sistema Linux, si en cambio estás usando Windows, para ejecutar Python solo debes escribir **python** en la terminal de comandos, sin agregar el número 3. Ten esto en cuenta en todo el documento a partir de este momento). El resultado debe ser algo similar a lo siguiente:



```
Python 3.10.12 (main, Jul 29 2024, 16:56:48) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Este mensaje nos muestra información sobre la versión de Python instalada, el sistema operativo en el que se instaló e información general. Inmediatamente después, aparece una línea con el símbolo `>>>`, el cual es conocido como el «Intérprete de Python» (en inglés: *Python prompt*). El intérprete de Python nos indica que podemos ingresar nuestros comandos desde ahí y serán ejecutados al presionar la tecla Enter.

Para nuestro primer ejemplo usaremos la función «`print`» definida por defecto en Python. Escribe el comando «`print("¡Hola mundo!")`» en tu sesión de Python y presionamos Enter. El resultado debería ser el siguiente:



```
>>> print("¡Hola mundo!")
¡Hola mundo!
>>>
```

Como puedes ver, la instrucción que se utilizó para mostrar el mensaje «¡Hola mundo!» es muy similar al lenguaje humano. Además, en la línea siguiente aparece de nuevo el intérprete de Python. Esto nos indica que Python está listo para seguir recibiendo instrucciones.

Para salir de Python y volver a una terminal de comandos normal, debes utilizar la función «`exit`» también definida por defecto. Específicamente, debes escribir el comando «`exit()`» y presionar Enter.


Esta es la forma más simple de ejecutar comandos de Python. Afortunadamente, no es la única. Para los propósitos del curso, necesitaremos de un Entorno de Desarrollo Integrado



(IDE, por sus siglas en inglés).

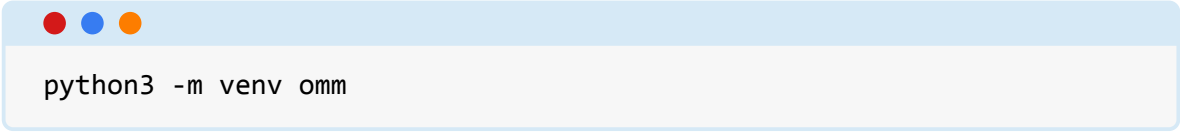
### 1.2.1. Creando un entorno virtual

Antes de continuar e independientemente de tu sistema operativo, es conveniente crear un entorno virtual de Python para evitar cualquier problema de compatibilidad con el sistema operativo y que las cosas funcionen correctamente. Primero debes abrir una nueva terminal de comandos e instalar el paquete `virtualenv`, ejecutando la instrucción:



```
pip install virtualenv
```

Cuando termine, escribe la siguiente instrucción para crear un entorno virtual llamado «omm» (puedes cambiar la palabra omm a cualquier otro nombre que prefieras, pero debe ser una única palabra, sin espacios):



```
python3 -m venv omm
```

El comando anterior creó un directorio llamado «omm» (o el nombre que hayas elegido) en el directorio en el que abriste la terminal de comandos. Por defecto, la terminal de comandos se abre en el directorio «C:\Users\user» en Windows, y en el directorio «/home/user/» en Linux/MacOSX (donde user es tu nombre de usuario en tu computadora, independientemente de tu sistema operativo).

Para activar el entorno virtual en Windows, debes ejecutar el siguiente comando:



```
C:\Users\user\omm\Scripts\activate
```

Ten en cuenta que debes cambiar la palabra user por tu nombre de usuario.

En cambio, para activarlo en Linux y Mac OSX, se hace con el comando:



```
source ~/omm/bin/activate
```

Como resultado, verás que a la izquierda de tu terminal de comandos aparece el nombre de tu entorno virtual encerrado en paréntesis, esto indica que está activado. Deberás ejecutar este comando para utilizar el entorno virtual de Python cada vez que abras una nueva terminal. Para desactivarlo, debes escribir el comando **deactivate** y ejecutarlo.

### 1.2.2. El editor Geany

Un programa de Python es simplemente un archivo de texto con extensión `.py` que puede escribirse en cualquier editor de texto. Existen muchos editores de texto y puedes usar el que prefieras. Recomiendo usar Geany porque es bastante simple, es ligero y fácil de instalar. Además permite ejecutar los programas directamente desde el editor, también utiliza el resaltado de sintaxis y permite usar una terminal integrada para ejecutar los códigos si así lo prefieres.

**Geany en Windows:** Para instalar Geany en Windows, dirígete a la página <https://www.geany.org/download/releases/> y descarga el instalador para Windows. Una vez que se haya descargado el archivo `.exe`, haz doble clic en él para iniciar el proceso de instalación. Sigue las instrucciones del asistente de instalación. Puedes dejar las opciones predeterminadas, a menos que desees personalizar la instalación.

**Configuración de Geany en Windows:** Una vez instalado, abre Geany desde el menú de aplicaciones. En la ventana del editor, escribe la instrucción `print("¡Hola mundo!")` y utiliza la combinación de teclas **Ctrl+S** para guardar el archivo. Puedes guardar el archivo con el nombre `hello_world.py` en tu carpeta de trabajo. Asegúrate de colocar la extensión `.py` en el nombre de tu archivo al momento de guardarlo.

Para que Geany funcione correctamente con el entorno virtual, se debe hacer una configuración sencilla. Dentro de la ventana de Geany, dirígete a **Build -> Set Build Commands**. Ahora haz lo siguiente:

1. En el campo llamado «Compile», edita su contenido para que el comando tenga lo siguiente:

```
C:\Users\user\omm\Scripts\python -m py_compile "%f"
```

2. En el campo llamado «Execute», edita su contenido para que el comando tenga lo siguiente:

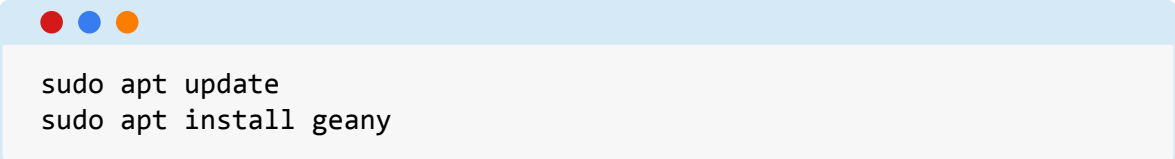
```
C:\Users\user\omm\Scripts\python "%f"
```

3. Haz clic en OK para guardar los cambios.

Usa las opciones de Build o los atajos de teclado (por ejemplo, F8 para compilar y F5 para ejecutar) para compilar y ejecutar el archivo Python. Verifica que el intérprete de Python utilizado sea el del entorno virtual y no el global.

El resultado debería ser un mensaje que dice «¡Hola mundo!», tal como cuando se ejecutó el comando desde una terminal.

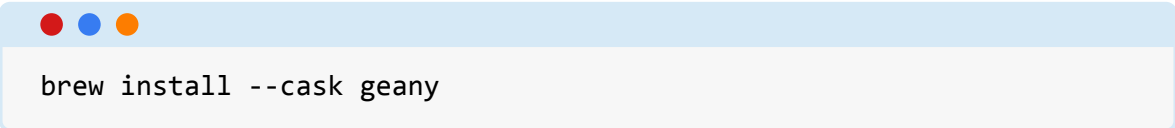
**Geany en Linux y Mac OSX:** Para instalar Geany en Linux, es suficiente con que ejecutes los siguientes comandos en una terminal:

A terminal window with a light blue header bar containing three colored circles (red, blue, orange). The terminal text is as follows:

```
sudo apt update
sudo apt install geany
```

```
sudo apt update
sudo apt install geany
```

En Mac OSX, puedes usar Homebrew para instalar Geany desde una terminal con el comando:

A terminal window with a light blue header bar containing three colored circles (red, blue, orange). The terminal text is as follows:

```
brew install --cask geany
```

```
brew install --cask geany
```

La configuración de Geany en Linux y Mac OSX es la misma. Para hacerlo, sigue los pasos descritos en la sección «Configuración de Geany en Windows» con la única diferencia que el campo «Compile» debe contener lo siguiente: `/home/user/omm/bin/python -m py_compile "%f"` y el campo «Execute» debe tener: `/home/user/omm/bin/python "%f"`.

Ten en cuenta que tanto en Windows como en Linux y Mac OSX, debes reemplazar la palabra «user» por tu nombre de usuario. Utilizaremos Geany principalmente para escribir y editar algunas funciones que utilizaremos. Ahora revisaremos algunos IDEs un poco más avanzados y que permiten hacer más cosas.

### 1.2.3. Jupyter Notebook

Jupyter Notebook es un IDE de Python basado en navegadores web, que además de ejecutar códigos de Python, permite incluir texto con formato, ecuaciones, imágenes y mucho más. Esencialmente, es un cuaderno donde es posible escribir código y tomar apuntes. Para instalarlo debes abrir una nueva terminal, activar el entorno virtual que creaste anteriormente y ejecutar la instrucción `pip install notebook`. En una terminal se vería así:

```
source ~/omm/bin/activate
pip install notebook
```

Recuerda utilizar los comandos adecuados si usas Windows. Una vez que termine la instalación, puedes usar Jupyter Notebook escribiendo la instrucción `jupyter notebook` (separado y en minúsculas) en la misma terminal de comandos. Esto abrirá una nueva pestaña en tu navegador predeterminado y te mostrará un explorador de archivos con el contenido de la carpeta donde ejecutaste la instrucción anterior, como se muestra en la Figura 1.1

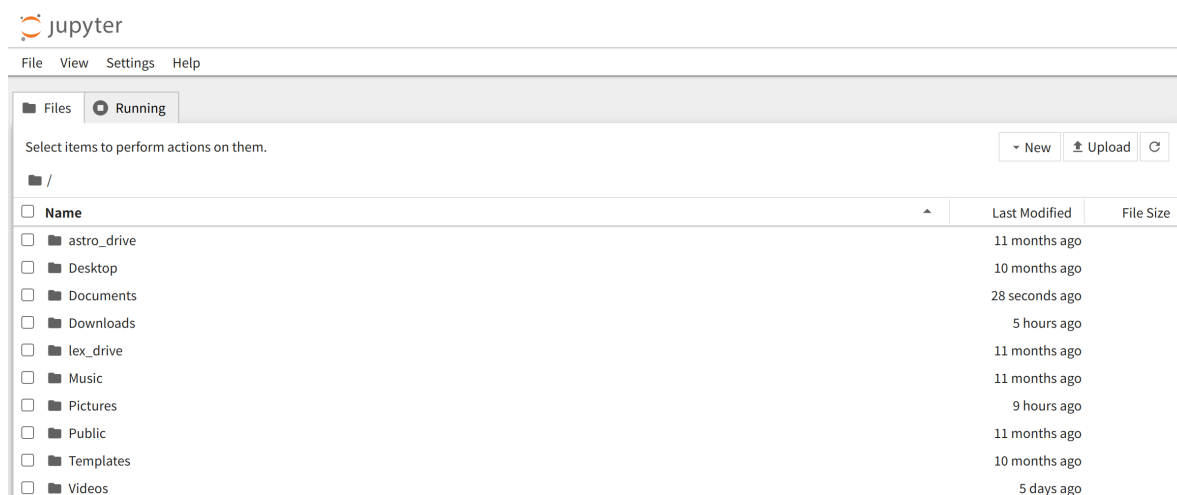


Figura 1.1: Interfaz gráfica de Jupyter Notebook

Desde la interfaz de Jupyter dirígete a tu carpeta de trabajo y crea un nuevo Notebook. Para lograrlo, haz click en el botón «New» en la esquina superior derecha y selecciona la opción «Notebook», luego elige el *kernel* llamado «Python 3 (ipykernel)» y esto abrirá una nueva pestaña en tu navegador.

En esa nueva pestaña, verás que aparecen los caracteres `[ ]`: (encerrado con un círculo rojo en el panel izquierdo de la Figura 1.2) y a la derecha aparece una celda vacía donde

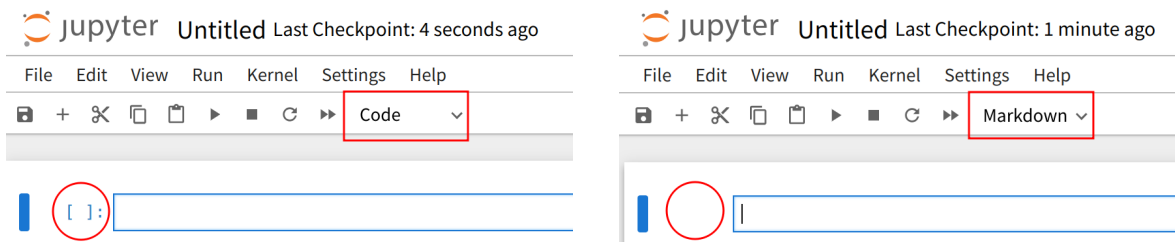


Figura 1.2: Opciones disponibles para escribir en Jupyter

puedes escribir los comandos de Python. Los caracteres `[ ]:` en Jupyter Notebook son el equivalente a los caracteres `>>>` en la terminal.

En la parte superior del Notebook hay una barra de menú, que a su vez contiene un botón con la palabra «Code» (encerrada con un rectángulo rojo en el panel izquierdo de la Figura 1.2). Al darle clic se muestra un menú desplegable cuyas opciones son «Code» y «Markdown». Este último es un tipo de *lenguaje de marcas* bastante popular para escribir texto con formato. Si seleccionas «Markdown», los símbolos `[ ]:` de la celda desaparecen (como se muestra en el panel derecho de la Figura 1.2) y ahora puedes escribir texto, ecuaciones, imágenes y más. Puedes revisar algunas nociones básicas sobre la escritura de texto Markdown en este enlace de [Github Docs](#).

Cuando vas a escribir código, siempre debes verificar que la celda está en modo «Code» o en su defecto, que a la izquierda de la celda aparezcan los símbolos `[ ]:`, de lo contrario, tus comandos no se ejecutarán. Por ejemplo, da clic en la celda vacía y asegúrate que se encuentre en modo «Code» y escribe el comando `print("¡Hola mundo!")`. Para ejecutar el código de una celda, debes dar click a la celda y usar la combinación de teclas **Shift+Enter**. El resultado debería ser algo similar a lo siguiente:

```
[1]: print("¡Hola mundo!")  
¡Hola mundo!
```

Para cerrar correctamente Jupyter Notebook, dirige tu cursor hacia la esquina superior izquierda del Notebook y selecciona `File -> Shutdown`. Una vez que lo hagas, podrás cerrar esa pestaña del navegador. Debes repetir este procedimiento para todas las pestañas del Notebook abiertas. Como último dato sobre Jupyter Notebook, debes saber que aunque necesite de un navegador web para funcionar, en realidad no necesita de una conexión a internet para ejecutar los comandos. El navegador web es únicamente para la interfaz gráfica.

#### 1.2.4. Visual Studio Code y PyCharm

Visual Studio Code (VS Code) y PyCharm son otros IDEs muy populares que también trabajan con Notebooks de Python y también archivos de texto plano con extensión `.py`. Su interfaz es similar a la de Jupyter Notebook y funcionan de la misma manera. Puedes revisar las instrucciones de instalación y configuración de VS Code y PyCharm en los sitios web oficiales <https://code.visualstudio.com/> y <https://www.jetbrains.com/es-es/pycharm/download/>, respectivamente.

Como última opción, si no te es posible instalar Python en tu computadora, puedes utilizar la opción en línea llamada Google Colaboratory (o simplemente Colab), a la que puedes acceder desde tu cuenta de google drive. La ventaja de usar Colab es que no necesitas instalar nada en tu computadora y puedes usar todos los paquetes de Python que requieras. Sin embargo, sí necesitarás de una conexión a internet para poder utilizarlo.

PyCharm, VS Code, Colab y Jupyter Notebook son totalmente equivalentes entre sí, teniendo diferencias mínimas y por lo tanto puedes utilizar el IDE que sea de tu preferencia. Teniendo esto en cuenta, podemos iniciar con algunas nociones sobre la escritura de código Python usando Notebooks.

### 1.3. Python como calculadora

#### 1.3.1. Operaciones aritméticas

Una de las formas elementales en las que Python puede ser utilizado es como una calculadora. En este sentido, la Tabla 1.1 muestra las operaciones aritméticas que pueden ejecutarse con Python. Quizá no estés muy familiarizado con las operaciones de división entera (representada con los caracteres `//`) y división modular (representada con el carácter `%`), pero las revisaremos con detalle más adelante.

Abre una nueva terminal de comandos, activa tu entorno virtual de Python e inicia una nueva sesión de Jupyter Notebook. Puedes abrir el mismo Notebook que creaste anteriormente o crear uno nuevo. Asegúrate que las celdas estén en modo «Code» e intenta ejecutar los siguientes comandos. El resultado de cada comando se mostrará al lado de los caracteres `[ ]:` (en color naranja).

Tabla 1.1: Operadores aritméticos

Símbolo/Operador	Significado/Operación
+	Suma
-	Resta
*	Multiplicación
/	División
**	Exponenciación
//	División entera
%	División modular

Veamos cómo funcionan los operadores aritméticos con unos ejemplos sencillos. Comenzamos con una simple, para calcular el valor de la operación  $5 + 9$ :

[2]: `5 + 9`

[2]: 14

La siguiente expresión es equivalente a  $40 - 5 \times 4$ :

[3]: `40 - 5*4`

[3]: 20

Y también podemos agrupar términos utilizando los paréntesis. Por ejemplo, podemos calcular el valor de la expresión  $(40 - 5 \times 4)/4$  usando la siguiente sintaxis:

[4]: `(40 - 5 * 4) / 4`

[4]: 5.0

Y para calcular el valor de  $3^2 + 4^2$  se usa la sintaxis:

[5]: `4**2 + 3**2`

[5]: 25

Ahora revisemos con un poco de detalle el operador `//`, que en Python es el operador de división entera. Este operador divide dos números y redondea el resultado hacia abajo al número entero más cercano. Por ejemplo, sabemos que  $7/3 = 2.333$ . Si aplicamos el operador de división entera a esos mismos números se obtiene:

[6]: `7 // 3`

[6]: 2

En este caso, `7 // 3` es igual a 2 porque la parte fraccionaria (0.333...) se descarta. Ahora veamos cómo se comporta este operador con números negativos:

[7]: `-7 // 3`

[7]: -3

En este caso, `-7 // 3` es igual a -3 porque el resultado se redondea hacia abajo (es decir, al entero más negativo).

Por otro lado, el operador `%` en Python es el operador de división modular (también conocido como resto o residuo). Este operador devuelve el resto de la división entre dos números. Es útil para determinar si un número es divisible por otro, o para obtener la parte sobrante después de dividir un número por otro. Veamos cómo funciona con ejemplos:

[8]: `7 % 3`

[8]: 1

En este caso, `7 % 3` es igual a 1 porque 7 dividido entre 3 es igual a 2 con un residuo de 1.

### 1.3.2. Operaciones matemáticas avanzadas

Para realizar operaciones matemáticas más avanzadas, necesitaremos utilizar un *módulo* de Python llamado `math`. Un módulo es un archivo que contiene definiciones y declaraciones de Python, como funciones, variables y clases. Los módulos permiten organizar y reutilizar código.

Para usar el módulo `math`, debemos importarlo en nuestro código. La forma más sencilla de hacerlo es usando la palabra clave «`import`». Dentro del módulo `math` están definidas muchas funciones matemáticas como la raíz cuadrada, logaritmos, funciones trigonométricas y mucho más. Para usarlas, debemos hacerlo con el operador de acceso, que en Python es un punto. La siguiente celda de código muestra cómo importar el módulo `math` y cómo acceder a la función raíz cuadrada, definida como `sqrt` dentro de dicho módulo:



```
[9]: import math  
  
     print(math.sqrt(16))
```

4.0

También podemos hacer importaciones específicas de las funciones dentro del módulo. Para esto usamos la palabra «**from**». El siguiente ejemplo muestra cómo importar la función `sqrt` definida dentro del módulo `math`:

```
[10]: from math import sqrt  
  
      print(sqrt(16))
```

4.0

En este último caso, no fue necesario escribir `math.sqrt` para calcular la raíz cuadrada, porque importamos directamente la función `sqrt`.

Adicionalmente, puedes importar módulos o funciones específicas y renombrarlas a tu gusto usando la palabra «**as**». Los siguientes dos ejemplos muestran cómo funciona esta sintaxis.

```
[11]: import math as m  
  
      print(m.sqrt(16))
```

4.0

```
[12]: from math import sqrt as raiz_cuadrada  
  
      print(raiz_cuadrada(16))
```

4.0

Como normal general, se recomienda hacer todas las importaciones al inicio del programa y utilizar alias claros y fácilmente identificables.

# Clase 2 | Variables y contenedores

## 2. Introducción

En cualquier lenguaje de programación, las variables actúan como nombres simbólicos que almacenan datos, permitiéndonos utilizar la información de manera eficiente a lo largo de nuestros programas. Los contenedores, por otro lado, son estructuras de datos que permiten agrupar múltiples valores en una sola entidad, facilitando la organización y gestión de conjuntos de datos. A lo largo de esta lección, aprenderemos cómo declarar variables, entenderemos los tipos de datos básicos y profundizaremos en los contenedores más utilizados en Python, como listas, tuplas, conjuntos y diccionarios.

### 2.1. Variables

Las variables son parte fundamental en cualquier lenguaje de programación y constan simplemente de dos partes: el nombre de la variable y su valor. Ambas partes están separadas por el signo de igual (=). Es decir, se escribe el nombre de la variable a la izquierda del símbolo = y su valor se escribe a la derecha. Si se quisiera por ejemplo definir una variable llamada `var_1` y asignarle el valor numérico 53, se haría de la siguiente manera:

```
[1]: var_1 = 53
```

Los nombres de las variables pueden contener letras minúsculas, mayúsculas, dígitos (0 - 9) y guiones bajos (\_). Sin embargo, los nombres no pueden empezar con dígitos; deben iniciar necesariamente con una letra o un guion bajo. Por ejemplo, los nombres «`_var`», «`_var1`» o «`_var_1`» son válidos, pero «`1_var`» y «`1var`» son nombres no permitidos y devuelven un error, como puedes verificar con el siguiente ejemplo:

```
[2]: 1var = 53
```

```
Cell In[1], line 1
1var = 53
^
SyntaxError: invalid decimal literal
```

El mensaje de error nos indica que estamos colocando un valor numérico en una posición inválida al momento de definir la variable.

### 2.1.1. Variables tipo int/float

En Python se distinguen dos tipos principales de variables numéricas: las variables tipo **int**, que corresponden a los números enteros y las variables tipo **float**, que corresponden a números decimales o muy grandes escritos con notación científica. Por ejemplo, la variable definida anteriormente y que llamamos `var_1` es de tipo **int**. En cambio, los valores numéricos 1.5 y  $4 \times 10^3$  son de tipo **float**. En realidad, la diferencia entre variables tipo **int** y **float** es más complicada que simplemente eso, pero no entraremos en detalles. En el siguiente ejemplo se definen dos variables tipo **float**.

```
[3]: var_2 = 1.5  
     var_3 = 4e3
```

Nota cómo la expresión  $4 \times 10^3$  se sustituye por `4e3`, haciendo muy fácil escribir expresiones con valores numéricos grandes. Las variables **int** y **float** admiten las operaciones aritméticas vistas en la Clase 1. Al realizar una operación aritmética entre un **int** y un **float**, el resultado será un **float**.

### 2.1.2. Variables de tipo string

Otro tipo de variables son las de tipo *string* (o cadena de caracteres), que se caracterizan porque son un conjunto de palabras o caracteres formando un mensaje, tales como «¡Hola mundo!». En Python, estas variables se identifican como de tipo **str** y para definir las se debe escribir el mensaje que queramos dentro de comillas simples (') o comillas dobles ("). Esto se muestra en el siguiente ejemplo:

```
[4]: message_1 = 'Hola'  
     message_2 = "y buenos días"
```

Es posible aplicar las operaciones aritméticas de suma y multiplicación a las variables **str**. Al aplicar la suma a las variables `message_1` y `message_2` se forma una nueva oración. Intenta ejecutar las siguientes dos celdas de código:

```
[5]: message_4 = message_1 + message_2
      print(message_4)
```

Holay buenos días

```
[6]: message_5 = message_1 + ' ' + message_2
      print(message_5)
```

Hola y buenos días

La diferencia entre las variables `message_4` y `message_5` es que a `message_5` se le agregó explícitamente un espacio en blanco ( ' ') al momento de sumar las variables `message_1` y `message_2`. Las variables `str` no pueden sumarse con variables tipo `int` ni `float`.

Por otro lado, la multiplicación de una variable `str` y una de tipo `int` sí está permitida. Intenta ejecutar lo siguiente:

```
[7]: message_1 * 3
```

```
[7]: HolaHolaHola
```

En resumen, se multiplicó `message_1` por el número `3`, y el mensaje apareció tres veces. Las variables `str` no pueden ser multiplicadas por variables tipo `float`.

### 2.1.3. Comentarios

Los comentarios son parte esencial de cualquier lenguaje de programación. Se trata de líneas de código que son ignorados por el intérprete/compilador y sirven para documentar el código. Mientras más líneas de código escribas, más difícil será recordar qué hace cada línea o bloque de código. Debido a esto se hacen necesarios los comentarios, y en Python hay dos formas de escribirlos. La primera es usando el símbolo de numeral: `#`. Cualquier cosa que escribamos después del símbolo será ignorada por Python:

```
[8]: pi = 3.14159265      #- Una variable float
      num = 5             #- Una variable int
      message = "nueve"   #- Una variable str
```

En la primera línea del ejemplo anterior, se declaró una variable llamada `pi` y se le asignó un valor de `3.14159265` y seguidamente se escribió un comentario para aclarar el tipo de variable que se está definiendo. Esto se repitió para las variables `num` y `message`.

La otra forma de escribir comentarios consisten en colocarlos dentro de triples comillas simples (`' ' ' ' '`) o triples comillas dobles (`""" """`). Las triples comillas permiten que el texto entre ellas abarque más de una sola línea. En realidad, estas son un tipo especial de comentarios que se utilizan para documentar las funciones, métodos, clases o módulos y por lo tanto son llamadas *docstrings* (cadenas de documentación).

#### 2.1.4. Variables tipo bool

Las variables tipo bool, o variables booleanas son aquellas que solo pueden tomar dos posibles valores: **True** (verdadero) o **False** (falso), a las cuales corresponden los valores numéricos de `0` y `1`, respectivamente. Este tipo de variables aparecen cuando se hacen comparaciones entre una o más variables. Para eso se utilizan operadores de comparación, de los cuales, los más comunes se muestran en la Tabla 2.1

Tabla 2.1: Operadores de comparación

Operador	Significado
<code>&gt;</code>	Mayor que
<code>&lt;</code>	Menor que
<code>&gt;=</code>	Mayor o igual que
<code>&lt;=</code>	Menor o igual que
<code>==</code>	Igual que
<code>!=</code>	Diferente de

Por ejemplo, sabemos que el número `3` es menor que el número `5`. Entonces la expresión `3 > 5` es falsa, como puedes comprobarlo:

```
[9]: 3 > 5
```

```
[9]: False
```

Estos operadores también pueden ser acompañados por las palabras clave definidas en Python: **and** y **or**. Intenta ejecutar los siguientes ejemplos:

```
[10]: 5 <= 2 and 4 > 2
```

```
[10]: False
```

Veamos con detalle qué es lo que está pasando en este ejemplo. El número `5` es mayor que `2`, por lo tanto, la expresión `5 <= 2` es falsa. Por otro lado, el número `4` es mayor que `2`, por lo tanto, la expresión `4 > 2` es verdadera. En otras palabras, la expresión es equivalente

a: **False and True**. El operador **and** devuelve un valor verdadero únicamente si todas las expresiones son verdaderas, de lo contrario devuelve un valor falso. Ya que en este caso una de las expresiones es falsa, entonces el resultado es **False**. Ahora revisa el siguiente ejemplo:

```
[11]: 5 <= 2 or 4 > 2
```

```
[11]: True
```

Estamos haciendo las mismas comparaciones que en el ejemplo anterior, pero ahora con el operador **or**, el cual devuelve un valor falso únicamente si todas las expresiones son falsas, de lo contrario devuelve un valor verdadero. En este caso, la expresión de la derecha es verdadera y por lo tanto, el resultado es un valor **True**.

Las variables booleanas se utilizan para iniciar, terminar o repetir algún proceso dependiendo si alguna condición se cumple o no. Esto lo veremos más adelante cuando trabajemos con controladores de flujo.

## 2.2. Contenedores

Los contenedores son un tipo de estructuras de datos que se utilizan para almacenar múltiples elementos o variables bajo un solo nombre. A diferencia de las variables, que pueden contener un único valor a la vez, los contenedores permiten mantener una colección de valores, los cuales a su vez pueden ser otros contenedores. Los contenedores son fundamentales para manejar datos de manera eficiente en Python, ya que permiten agrupar, iterar, acceder y manipular conjuntos de valores de manera ordenada y flexible. Los contenedores disponibles en python son las listas, tuplas, conjuntos y diccionarios.

### 2.2.1. Listas

Las listas son secuencias ordenadas de elementos o variables de cualquier tipo, que no tienen que estar relacionadas entre sí. Para definir una lista se colocan los elementos separados por comas y encerrados entre corchetes. El siguiente ejemplo muestra que una lista puede contener cualquier tipo de variable:

```
[12]: lista_1 = [1, 'Uno', True]
      print(lista_1)
```

```
[1, 'Uno', True]
```

También es posible que una lista contenga a otras listas:

```
[13]: lista_2 = [lista_1, [0, 'Cero', False]]  
[[1, 'Uno', True], [0, 'Cero', False]]
```

Además, el operador numérico de la suma está permitido en las listas. El resultado es una nueva lista:

```
[14]: lista_3 = lista_1 + lista_2  
print(lista_3)  
[1, 'Uno', True, [1, 'Uno', True], [0, 'Cero', False]]
```

Como se mencionó, las listas son una colección ordenada de elementos. Debido a esta característica, sus elementos pueden ser accedidos mediante índices, que corresponden a la posición del elemento en la lista. Sin embargo, se debe tener en cuenta que Python es un lenguaje de programación que comienza a contar desde el número cero. Es decir, al primer elemento le corresponde la posición (o índice) `0`, al segundo elemento la posición `1` y así sucesivamente. Por ejemplo, definamos una nueva lista con los nombres de algunos astrónomos:

```
[15]: astronomers = ['Carl Sagan', 'Stephen Hawking', 'Jocelyn Bell']
```

Para acceder a un elemento de la lista `astronomers`, se escribe el nombre de la lista y se acompaña del índice encerrado entre corchetes y sin espacios. Específicamente, para acceder a `'Carl Sagan'`, que es el primer elemento, lo hacemos de la siguiente manera:

```
[16]: astronomers[0] # Selecciona el primer elemento
```

```
[16]: 'Carl Sagan'
```

El último elemento de cualquier lista, independiente de cuántos elementos tenga, puede ser accedido mediante el índice `-1`, como puedes verificar:

```
[17]: astronomers[-1] # Selecciona el último elemento
```

```
[17]: 'Jocelyn Bell'
```

Una de las características de las listas es que sus elementos pueden ser modificados.

En un contexto de programación, esto significa que las listas son *mutables*. Por ejemplo, para reemplazar el primer elemento de la lista por el nombre algún otro astrónomo, podemos hacerlo de la siguiente manera:

```
[18]: astronomers[0] = 'Johannes Kepler'
      print(astronomers)

['Johannes Kepler', 'Stephen Hawking', 'Jocelyn Bell']
```

Puedes agregar elementos a una lista usando el método `list.append()` (el cual permite agregar un elemento a la vez) y el método `list.extend()` (el cual permite agregar varios elementos a la vez, en formato de lista). Ambos métodos agregan los elementos al final de la lista. Por ejemplo, para agregar nuevamente a Carl Sagan a la lista de astrónomos, podemos hacerlo de la siguiente manera:

```
[19]: astronomers.append('Carl Sagan')
      print(astronomers)

['Johannes Kepler', 'Stephen Hawking', 'Jocelyn Bell', 'Carl Sagan']
```

Podemos en cambio, agregar dos elementos al mismo tiempo de la siguiente manera:

```
[20]: astronomers.extend(['Henrietta Leavitt', 'Vera Rubin'])
      print(astronomers)

['Johannes Kepler', 'Stephen Hawking', 'Jocelyn Bell', 'Carl Sagan',
 'Henrietta Leavitt', 'Vera Rubin']
```

Algunos tipos de variables permiten ser convertidos a listas, como es el caso de las variables de tipo `str`. Para lograrlo, se utiliza la función `list()` predefinida en Python. Revisa el siguiente ejemplo:

```
[21]: letters = list(astronomers[2])
      print(letters)

['J', 'o', 'c', 'e', 'l', 'y', 'n', ' ', 'B', 'e', 'l', 'l']
```

Adicionalmente, las variables de tipo `str` también admiten acceder a sus elementos mediante índices:

```
[22]: name = 'Jocelyn'
      name[0]
```

```
[22]: 'J'
```



También puedes definir listas que sean una porción de otra lista. A esto se le conoce como «*rebanar*» o en inglés, «*slice*». Supongamos por ejemplo, que solo queremos seleccionar los primeros 3 elementos de la lista `astronomers`, entonces usamos la siguiente sintaxis:

```
[23]: new_astronomers = astronomers[0:3]
      print(new_astronomers)

['Johannes Kepler', 'Stephen Hawking', 'Jocelyn Bell']
```

El ejemplo anterior está seleccionando los elementos de la lista que van desde el índice 0 (primer elemento) hasta el índice 3 (cuarto elemento). Por defecto en Python, el índice después de los dos puntos (:) no se incluye, por eso la lista `new_astronomers` solo tiene tres elementos y no cuatro.

### 2.2.2. Tuplas

Las tuplas representan la versión inmutable de las listas, es decir, pueden almacenar cualquier tipo de elementos pero sus valores no se pueden modificar. Para definir las, basta con escribir los elementos separados por comas sin encerrarlos:

```
[24]: planets_1 = 'Mercurio', 'Venus', 'Tierra'
      planets_1
```

```
[24]: ('Mercurio', 'Venus', 'Tierra')
```

Alternativamente, las tuplas se definen al escribir los elementos separados por comas y encerrarlos entre paréntesis:

```
[25]: planets_2 = ('Marte', 'Júpiter', 'Saturno')
      planets_2
```

```
[25]: ('Marte', 'Júpiter', 'Saturno')
```

Al igual que en el caso de las listas, se puede acceder a los elementos de una tupla mediante índices y se puede convertir una variable de tipo `str` a una tupla usando la función `tuple()`. usando esa misma función, también puedes convertir una lista en una tupla.

### 2.2.3. Conjuntos

Los conjuntos (en inglés llamados «sets») son colecciones de datos no ordenados y además no permiten elementos duplicados. Esto significa que no es posible acceder a sus elementos mediante índices y que ninguno de sus elementos se repite. Los conjuntos se definen con sus elementos separados por comas y encerrados entre llaves:

```
[26]: my_set = {1,2,3,4,5,5,5}
      print(my_set)
```

```
{1, 2, 3, 4, 5}
```

También puedes definirlos con la función `set()`:

```
[27]: my_set = set([1,2,3,4,5,5,5])
```

```
{1, 2, 3, 4, 5}
```

Los conjuntos también son contenedores mutables. Para añadir elementos se utiliza la función `add()`:

```
[28]: my_set.add(6)
      print(my_set)
```

```
{1, 2, 3, 4, 5, 6}
```

Para eliminar un elemento se utilizan los métodos `remove()` o `discard()`, como puedes comprobar:

```
[29]: my_set.remove(3)
      print(my_set)
```

```
{1, 2, 4, 5, 6}
```

```
[30]: my_set.discard(2)
      print(my_set)
```

```
{1, 4, 5, 6}
```

La diferencia en ambos métodos es que `remove()` lanzará un error si el elemento no existe, mientras que `discard()` no hará nada si el elemento no está presente.

También puedes usar el operador `in` para verificar si un elemento está en el conjunto. Ten en cuenta que este operador también puede utilizarse para verificar si un elemento está en una lista o en una tupla. Veamos un ejemplo solo para los conjuntos:

```
[31]: print(2 in my_set)
```

```
False
```

Otras operaciones disponibles para los conjuntos son:

- **Unión:** combina todos los elementos de dos conjuntos, eliminando los duplicados, mediante el operador «|».
- **Intersección:** Devuelve los elementos comunes en ambos conjuntos, mediante el operador «&».
- **Diferencia:** Devuelve los elementos que están en el primer elemento pero no en el segundo, mediante el operador «-».
- **Diferencia simétrica:** Devuelve los elementos que están en un set o en otro, pero no en ambos, mediante el operador «^».

#### 2.2.4. Diccionarios

Los diccionarios son estructuras de datos no ordenados y están compuestos de pares *clave-valor* (en inglés *key-values*), donde cada valor está asociado a una clave única. Para crear un diccionario se escriben los pares de clave-valor separados por comas y encerrados con llaves {}, o encerrados por la función `dict()`. Para entenderlo mejor, revisa el siguiente ejemplo:

```
[32]: stephen = {  
    "first_name": "Stephen",  
    "last_name": "Hawking",  
    "birthday": (8, 1, 1942)  
}  
print(stephen)
```

```
{'first_name': 'Stephen', 'last_name': 'Hawking', 'birthday': (8, 1,  
1942)}
```

```
[33]: albert = dict(  
    first_name="Albert",  
    last_name='Einstein',  
    birthday= (14, 3, 1879)  
)  
print(albert)
```

```
{'first_name': 'Albert', 'last_name': 'Einstein', 'birthday': (14, 3,  
1879)}
```

Las dos sintaxis anteriores son totalmente equivalentes para definir un diccionario. Para acceder a cualquier valor de un diccionario, se escribe el nombre del diccionario y en seguida, encerrado entre corchetes se escribe su clave correspondiente. Por ejemplo:

```
[34]: stephen['first_name']
```

```
[34]: 'Stephen'
```

Los diccionarios son contenedores mutables, por lo tanto sus valores pueden cambiar, pero no sus claves. Puedes agregar un par clave-valor simplemente asignando un valor a una nueva clave:

```
[35]: stephen['nationality'] = "British"
      print(stephen)
```

```
{'first_name': 'Stephen', 'last_name': 'Hawking', 'birthday': (8, 1,
1942), 'nationality': 'British'}
```

Puedes obtener todas las claves, valores, o pares clave-valor usando los métodos `keys()`, `values()` y `items()`, respectivamente:

```
[36]: albert.keys()
```

```
[36]: dict_keys(['first_name', 'last_name', 'birthday'])
```

```
[37]: albert.values()
```

```
[37]: dict_values(['Albert', 'Einstein', (14, 3, 1879)])
```

```
[38]: albert.items()
```

```
[38]: dict_items([('first_name', 'Albert'), ('last_name', 'Einstein'),
('birthday', (14, 3, 1879))])
```

# Clase 3 | Control de flujo y lógica

## 3. Controles de flujo

Los controles de flujo en un lenguaje de programación permiten que ciertas partes de un código se ejecuten y otras no, dependiendo de si se cumplen o no algunas condiciones. Adicionalmente, también permiten ejecutar líneas de código una y otra vez mientras alguna condición siga siendo válida. Debido a esto, los principales controles de flujo y lógica son llamados *condicionales* y *bucles* (en inglés, *loops*). En esta clase revisaremos ambos conceptos.

### 3.1. Condicionales

Son el tipo de controladores de flujo más sencillos, su funcionamiento puede describirse de la siguiente manera: "Si la variable  $x$  es verdadera, entonces realiza una tarea; en caso contrario, realiza esta otra tarea". Para escribir este tipo de condiciones se utilizan las palabras en inglés «if» y «else», que significan «si» y «en caso contrario», respectivamente.

#### 3.1.1. Declaraciones if

La forma más sencilla de usar los condicionales es cuando solo se quiere comprobar si alguna condición es verdadera y no hacer nada si la condición es falsa. En ese caso, se utiliza la palabra clave **if** una única vez. La sintaxis es la siguiente:

```
if <condition>:  
    <if-block>
```

Para este tipo de declaraciones, la condición, identificada como `<condition>` puede ser cualquier tipo de expresión de comparación o lógica que devuelva un valor **True** o **False**. Además, inmediatamente después de dicha condición se escriben dos puntos (:). El segmento `<if-block>` representa aquellos comandos o instrucciones que se ejecutarán únicamente si la condición es verdadera.

Algo muy importante es que los comandos del `<if-block>` deberán ser escritos abajo del «`if <condition>:`» con una sangría (en inglés «*indent*») de cuatro espacios. En Python, esta sangría es obligatoria siempre que se quiera separar bloques de código del resto de líneas de código. De no usar la sangría de cuatro espacios en los condicionales, el código generará un error. Como ejemplo, intenta ejecutar la siguiente celda de código:

```
[1]: #- Definiendo un número
      number = 0
      if number == 0.0:
      print(f"El número {number} es igual a 0.0")
```

```
Cell In[1], line 3
    print(f"El número {number} es igual a 0.0")
    ^
IndentationError: expected an indented block after 'if' statement on
line 2
```

Este error es bastante simple. Python nos está diciendo que se esperaba un bloque con sangría después de la declaración `if`. Puedes corregirlo simplemente agregando la sangría de cuatro espacios:

```
[2]: #- Definiendo un número
      number = 0
      if number == 0.0:
          print(f"El número {number} es igual a 0.0")
```

```
[2]: El número 0 es igual a 0.0
```

La condición `number == 0.0` es verdadera y por lo tanto se mostró el mensaje. Si la condición hubiera sido falsa, no se habría mostrado ningún mensaje. Nota cómo se definió a la variable `number` como de tipo `int` y en la condición se comparó con una variable de tipo `float`, de modo que la condición sería equivalente a `0 == 0.0`. Esta condición resulta ser cierta porque el operador `==` únicamente compara si dos valores son equivalentes.

### 3.1.2. Declaraciones if-else

Cada bloque `if` puede ir seguido de un bloque opcional `else`, que se ejecutará únicamente si la primera condición es falsa. El bloque `else` no necesita de ninguna condición pero sí de los dos puntos (`:`) y la línea siguiente también debe tener sangría de cuatro espacios. La sintaxis para este tipo de declaraciones es la siguiente:

```
if <condition>:
    <if-block>

else:
    <else-block>
```

Por ejemplo, podemos escribir un código que verifique si un número es par o impar, comprobando si el número es divisible por 2. Esto se puede hacer con una declaración **if-else** de la siguiente manera:

```
[3]: #- Definiendo un número
number = 15

#- Comprobando si es par o impar
if number % 2 == 0:
    result = "par"
else:
    result = "impar"

# Muestra un mensaje con el resultado
print(f"El número {number} es {result}.")
```

El número 15 es impar.

El número 15 no es divisible entre dos, por lo tanto la condición `number % 2 == 0` es falsa. Como resultado el bloque `result = "par"` no se ejecuta. En cambio, se ejecuta el bloque `result = "impar"` y se muestra el mensaje "El número 15 es impar".

### 3.1.3. Declaraciones if-elif-else

En muchas ocasiones será necesario comprobar más de dos posibles situaciones, y para esto se utiliza la sintaxis **if-elif-else**. La palabra *elif* es una abreviación de «*else if*» en inglés y en las declaraciones pueden haber cuantos bloques **elif** sean necesarios. El primer bloque que devuelva un valor **True** será ejecutado, y ninguno de los restantes será comprobado ni ejecutado. La sintaxis de este tipo de declaraciones es la siguiente:

```
if <condition>:
    <if-block>

elif <condition1>:
    <elif-block1>
```

```
elif <condition2>:
    <elif-block2>

else:
    <else-block>
```

El siguiente ejemplo muestra cómo usar las declaraciones **if-elif-else** para clasificar a una partícula fundamental de acuerdo a su masa:

```
[4]: #- Masa de La partícula hipotética
mass = 5.0 # en GeV/c^2

#- Clasificación basada en su masa
if mass < 0.1:
    classification = "Mesón Ligero"

elif 0.1 <= mass < 1.0:
    classification = "Mesón Pesado"

elif 1.0 <= mass < 10.0:
    classification = "Barión"

else:
    classification = "Partícula Exótica"

#- Mostrar su clasificación
print(f"La partícula con masa {mass} GeV/c^2 es un {classification}.")
```

[4]: La partícula con masa 5.0 GeV/c^2 es un Barión.

Nuevamente, ten en cuenta que en cada bloque **if-elif-else** se necesita colocar una sangría obligatoria.

### 3.1.4. Expresiones if-else

Para finalizar con los condicionales, es posible aplicar la sintaxis **if-else** en una simple expresión de una línea (sin los bloques de sangría). Se necesita de tres elementos y por lo tanto se le conoce como un operador condicional ternario. La sintaxis es la siguiente:

```
x if <condition> else y
```

Si la condición es verdadera, entonces el resultado de la expresión anterior es x, si es



falsa, el resultado es y. Esta expresión resulta muy útil para asignar valores a una variable con base en alguna condición. Esto se verifica en el siguiente ejemplo.

```
[5]: #-Definiendo un número
      number = -5

      #- Comprobar si es positivo o negativo y guardar el resultado
      result = "positivo" if number > 0 else "negativo"

      #- Mostrar el resultado
      print(f"El número {number} es {result}.")
```

[5]: El número -5 es negativo.

En este ejemplo, el operador ternario verifica si la variable `number` es mayor que cero. Si de hecho lo es, le asigna el valor `"positivo"` a la variable `result`, en caso contrario le asigna el valor `"negativo"`.

## 3.2. Bucles

Como se ha visto, los condicionales permiten que los bloques de código con sangría se ejecuten una única vez dependiendo de alguna condición. Los bucles, en cambio, permiten que el mismo bloque se ejecute muchas veces. Los tipos de bucles disponibles en Python son el bucle `while`, el bucle `for` y algo que es conocido como «contenedores por comprensión».

### 3.2.1. Bucle while

Los bucles `while` se relacionan con los condicionales porque permiten que el bloque de código se siga ejecutando mientras alguna condición sea verdadera. Su sintaxis es muy similar a la de las declaraciones `if`:

```
while <condition>:
    <while-block>
```

Para este tipo de declaraciones también se deben aplicar los dos puntos luego de la condición y el bloque de instrucciones también debe tener sangría. La condición es evaluada al inicio de cada iteración y si es verdadera, el bloque se ejecuta. Si la condición es falsa, el bloque es ignorado y el programa continúa.

El ejemplo más simple usando un bucle `while` consiste en generar una cuenta regresiva comenzando en un número entero arbitrario:

```
[6]: #- Número inicial
count = 5

#- Cuenta regresiva
while count > 0:
    print(count)
    count -= 1

print("¡Fin!") # Se muestra cuando el bucle termina
```

5  
4  
3  
2  
1  
¡Fin!

En este ejemplo particular, sucede lo siguiente:

- Se inicia con la variable `count` igual a 5
- El bucle `while` se ejecuta siempre y cuando `count` sea mayor que cero
- En cada iteración se imprime el valor actual de `count` y posteriormente su valor disminuye en una unidad
- Eventualmente, la variable `count` toma el valor de 0 y la condición se vuelve falsa
- Cuando el bucle termina, se imprime el mensaje "`¡Fin!`"

Es muy común cometer errores en los bucles `while`, el más frecuente es iniciar por accidente un bucle infinito. Esto sucede cuando la condición evaluada es siempre verdadera. Si en el ejemplo anterior no hubiésemos escrito la instrucción `count -= 1`, la variable `count` hubiera tenido siempre un valor igual a 5 y por lo tanto el bucle nunca habría finalizado. Al iniciar un bucle `while` debemos asegurarnos de agregar una línea de código que haga que la condición sea falsa en algún momento. Si por algún motivo inicias un bucle infinito, siempre puedes detenerlo con la combinación de teclas **Ctrl+C**.

También es posible terminar con el bucle `while` en cualquier momento, haciendo uso de la instrucción `break`. Por ejemplo, si quisiéramos encontrar el primer número par de una lista, podríamos hacerlo con el bucle `while` de la siguiente manera:

```
[7]: #- Lista de números
      numbers = [3, 7, 12, 5, 8, 10, 15]

      #- Variable para guardar el número par
      first_even = None

      #- Bucle while
      index = 0
      while index < len(numbers):
          if numbers[index] % 2 == 0:
              first_even = numbers[index]
              break # El bucle termina al encontrar el primer número par
          index += 1

      #- Imprime el valor encontrado
      if first_even is not None:
          print(f"El primer número par encontrado fue {first_even}.")
      else:
          print("No hay números pares en la lista.")
```

[7]: El primer número par encontrado fue 12.

### 3.2.2. Bucle for

Aunque los bucles **while** son bastante útiles para repetir declaraciones, en general resulta de mayor utilidad el iterar sobre un contenedor (como las listas, tuplas, conjuntos y diccionarios) o algún otro tipo de variable iterable (como las cadenas de caracteres). En estos casos, se toma un elemento del contenedor en cada iteración y el bucle termina cuando ya no hay más elementos. Esto se logra usando un bucle **for**. La sintaxis para los bucles **for** es diferente a la del bucle **while**, ya que no depende de ninguna condición y consta de lo siguiente:

```
for <variable> in <iterable>:
    <for-block>
```

De nuevo, necesitamos escribir los dos puntos para especificar cuándo inicia el bloque de código que se va a repetir, que a su vez debe tener sangría. En la sintaxis anterior, **<variable>** es el nombre de una variable que se asigna a un elemento cada vez que se ejecuta el bucle. El **<iterable>** es cualquier objeto que pueda devolver elementos. Todos los tipos de contenedores vistos en la Clase 2 son iterables. La cuenta regresiva puede reescribirse de la siguiente manera usando el bucle **for**:

```
[8]: for count in [5,4,3,2,1]:  
      print(count)  
  
      print("¡Fin!")
```

```
5  
4  
3  
2  
1  
¡Fin!
```

La instrucción **break** también se puede usar con los bucles **for**, y funcionan de la misma manera. Adicionalmente, también es posible utilizar la instrucción **continue**, que ignora el bloque de código únicamente en la iteración actual y luego continua con la siguiente. Por ejemplo, el siguiente ejemplo muestra cómo ignorar los números impares en una lista:

```
[9]: for num in [1, 2, 3, 4, 5, 6, 7, 8, 9]:  
      if num % 2 != 0:  
          continue # Ignorar y continuar con la siguiente iteración  
      print(num)  
  
      print("¡Listo!")
```

```
2  
4  
6  
8  
¡Listo!
```

La instrucción **continue** también se puede utilizar en los bucles **while**.

Para usar un bucle **for** con una cadena de caracteres se procede de la misma forma que con las listas. El resultado es que se itera por cada una de las letras, por ejemplo:

```
[10]: for letter in "Hola":  
       print(letter)
```

```
H  
o  
l  
a
```

También es posible aplicar un bucle **for** en conjuntos y diccionarios, pero recuerda que este tipo de contenedores no está ordenado, así que el resultado podría no verse como esperarías. Para ilustrar esto se muestra el siguiente ejemplo:

```
[11]: # Iteración sobre un conjunto
      for key in {'a', 'b', 'c', 'd', 'e'}:
          print(key)
```

```
b
c
f
d
a
e
```

El orden en que se muestra el resultado no es necesariamente al orden en el que se escribieron las letras "a", "b", "c", "d", "e" y "f". Como último ejemplo en esta sección, se muestra cómo puede usarse el ciclo **for** para iterar sobre los pares de clave-valor de un diccionario.

```
[12]: #- Definiendo el diccionario
      d = {'a': 1, 'b': 2, 'c': 3}
```

```
[13]: #- Iterar sobre las claves
      print("Keys:")
      for key in d.keys():
          print(key)
      print("=====")
```

```
Keys:
a
b
c
=====
```

```
[14]: #- Iterar sobre los valores
      print("Values:")
      for value in d.values():
          print(value)
      print("=====")
```

```
Values:
1
2
3
=====
```

```
[15]: #Iterar sobre los pares clave-valor
      print("Items")
      for key, value in d.items():
          print(f"Key: {key}, Value: {value}")
```

```
Items
Key: a, Value: 1
Key: b, Value: 2
Key: c, Value: 3
```

### 3.2.3. Contenedores por comprensión

Como se ha visto, los ciclos **while** y **for** permiten hacer muchas cosas, pero para usarlos se necesita de al menos dos líneas de código: una para definir el bucle y la otra para especificar las acciones. Por ejemplo, supongamos que tenemos una lista con los nombres de los primeros cuatro planetas escritos en minúscula, y también tenemos una lista vacía en la que deseamos agregar esos mismos nombres pero en mayúscula:

```
[16]: #- Lista con planetas
      planetas = ['mercurio', 'venus', 'tierra', 'marte']

      #- Lista vacía
      planetas_mayus = []
```

Para lograrlo podemos aplicar el método `list.upper()` a cada elemento de la lista `planetas` y luego anexarlos a la lista `planetas_mayus` en un bucle **for**:

```
[17]: for planet in planetas:
      planetas_mayus.append(planet.upper())

      print(planetas_mayus)

      ['MERCURIO', 'VENUS', 'TIERRA', 'MARTE']
```

Lo anterior puede realizarse de manera equivalente en una sola línea gracias a la sintaxis de contenedores por comprensión, que puede aplicarse tanto a listas como a diccionarios. La sintaxis en cada caso es:

```
# Lista por comprensión
[<expr> for <loop-var> in <iterable>]

# Diccionario por comprensión
{<key-expr>: <value-expr> for <loop-var> in <iterable>}
```

Así, volviendo al ejemplo de los planetas, se puede crear la lista de la siguiente manera:

```
[18]: planetas_mayus = [planet.upper() for planet in planetas]
```

Para el caso de un diccionario, supongamos que tenemos una lista de números y queremos crear una nueva lista con el cuadrado de esos números, entonces:

```
[19]: #- Definiendo una lista de números
      numbers = [1, 10, 12.5, 65, 88]

      #- Creando un diccionario por comprensión
      results = {x: x**2 for x in numbers}

      #- Mostrar el resultado
      print(results)

{1: 1, 10: 100, 12.5: 156.25, 65: 4225, 88: 7744}
```

También es posible realizar filtros a los contenedores por comprensión, con ayuda de condicionales. La sintaxis es la siguiente:

```
[20]: # Lista por comprensión con filtro
      [<expr> for <loop-var> in <iterable> if <condition>]

      # Diccionario por comprensión con filtro
      {<key-expr>: <value-expr> for <loop-var> in <iterable> if <condition>}
```

En este caso, si la condición resulta ser verdadera, entonces se agrega el elemento actual, en caso contrario se ignora y se evalúa el siguiente iterable. En este último ejemplo, se creará una lista con los planetas escritos en mayúscula si el nombre inicia con «m», y se calcularán los cuadrados de los números si el número es par:

```
[21]: #- Planetas que inician con M
      new_list = [planet.upper() for planet in planetas if planet[0] == 'm']
      print(new_list)

['MERCURIO', 'MARTE']
```

```
[22]: #- Cuadrado de números pares
      results_even = {x: x**2 for x in numbers if x%2 == 0}
      print(results_even)

{10: 100, 88: 7744}
```

# Clase 4 | Funciones

## 4. Funciones en Python

Una función en Python es un bloque de declaraciones (comandos) que realizan una tarea específica. Es posible que dicha tarea necesite ejecutarse en repetidas ocasiones para diferentes valores de entrada en un mismo código. El propósito de las funciones es, que en lugar de escribir el mismo código múltiples veces para diferentes entradas, podamos realizar llamadas a funciones y así reutilizar el código contenido en ella una y otra vez. En esta clase aprenderás a escribir funciones y comprenderás por qué utilizarlas hace que los programas sean más fáciles de escribir, leer y corregir.

### 4.1. Funciones predefinidas

En Python existen funciones predefinidas, las cuales están disponibles para cualquier usuario. La más común y quizás la que más hemos utilizado a lo largo de este curso es la función `print()`. En este [enlace](#) puedes ver la lista completa de las funciones predefinidas en Python y recomiendo que revises qué es lo que hacen y cómo se utilizan.

Tomemos como ejemplo la función predefinida llamada `input()`, que se utiliza para pedir al usuario que ingrese cualquier información que se le pida:

```
[1]: input('Ingresa tu nombre: ')
```

```
Ingresa tu nombre: Python
```

```
[1]: 'Python'
```

En el ejemplo anterior, se ingresó la palabra «Python» y el resultado fue esa misma cadena de caracteres. Debes tener cuidado, porque la función `input()` siempre asume que lo que ingresas es una cadena de caracteres, incluso si lo que ingresas es un número:

```
[2]: number = input('Ingresa un número: ')
      print(number)
      print(type(number))
```



```
[2]: Ingresar un número: 5
5
<class 'str'>
```

En el ejemplo anterior, se ingresó el número 5, pero al utilizar la función `type()` vemos que se guardó como una variable de tipo `str` y no como de tipo `int`.

Para que el número sea almacenado como de tipo `int`, se debe usar explícitamente la función `int()` en conjunto con `input()` de la siguiente manera:

```
[3]: number = int( input('Ingresar un número: ') )
print(number)
print(type(number))
```

```
[3]: Ingresar un número: 5
5
<class 'int'>
```

Como puedes darte cuenta, las funciones son parte fundamental de cualquier código ya que permiten realizar tareas específicas. Ahora veremos cómo crear nuestras propias funciones para realizar cualquier tarea que se nos ocurra.

## 4.2. Funciones definidas por el usuario

La estructura y sintaxis para declarar una función en Python se muestra en la Figura 4.1.

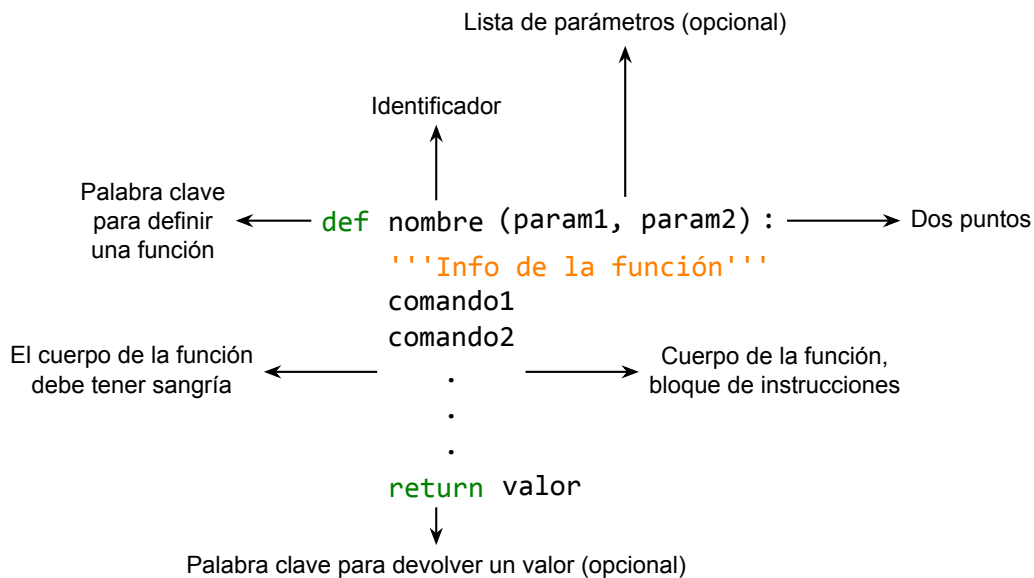


Figura 4.1: Estructura de una función en Python

Para declarar una función, primero se utiliza la palabra clave «**def**», que le informa a Python que se está definiendo una función. Luego de la palabra **def** y separado por un espacio, se debe escribir el nombre de la función, que sirve como *identificador* de la función. Inmediatamente después y sin espacios se colocan paréntesis y entre ellos se escriben los parámetros de la función separados por comas. Los parámetros son opcionales, como veremos más adelante. Al cerrar los paréntesis se escriben dos puntos. Esta primera línea constituye la «*definición de la función*».

Cualquier línea con sangría de cuatro espacios que se encuentre abajo de la definición de la función constituye el cuerpo de la función. El texto inmediatamente abajo de la definición, que en la figura dice `'''Info de La función'''`, constituye la documentación, o *docstring*, que describe qué es lo que hace la función. La documentación siempre se escribe entre comillas triples.

Las líneas debajo de la documentación son las líneas de código reales en el cuerpo de la función. En el ejemplo, estas líneas dicen `command1`, `command2`, ... y le indican a la función qué tarea debe realizar.

Si en el cuerpo de la función se calculó una variable llamada `valor`, dicha variable puede ser devuelta por la función en la última línea con ayuda de la palabra clave «**return**».

A continuación revisaremos diferentes tipos de funciones, comenzando por las más simples hasta llegar a declaraciones más estructuradas.

#### 4.2.1. Funciones sin parámetros

El ejemplo más sencillo es el de una función que no requiere de parámetros de entrada y que imprime un saludo. Veamos cómo definirla siguiendo la sintaxis de la Figura 4.1:

```
[4]: def salu2():  
      """Muestra un mensaje"""  
      print("¡Hola!")
```

Para utilizarla, la llamamos mediante su nombre:

```
[5]: #- Invocando La función  
      salu2()
```

```
[5]: ¡Hola!
```

Como la función no necesita de ningún parámetro, fue suficiente con escribir `salu2()` y no escribir nada más. Ahora veamos un poco sobre funciones que sí requieren de parámetros de entrada.

#### 4.2.2. Funciones con parámetros

La siguiente función requiere de un parámetro de entrada y muestra un mensaje personalizado cada vez que se invoca. Debes tener en cuenta que en este caso en particular, el parámetro de entrada tiene que ser una variable de tipo `str`.

```
[6]: def salu3(name):  
      """Muestra un mensaje personalizado"""  
      print(f"¡Hola, {name}!")
```

Para utilizarla, debemos llamarla mediante su nombre pero esta vez necesitamos agregar un *argumento*:

```
[7]: salu3("Señor Doctor Profesor Patricio")
```

```
[7]: ¡Hola, Señor Doctor Profesor Patricio!
```

Vale la pena hacer una aclaración: la variable «name» en la definición de la función `salu3()` es un parámetro, una pieza de información que la función necesita para hacer su trabajo. En cambio, el valor `"Señor Profesor Patricio"` en la expresión `salu3("Señor Profesor Patricio")` es un argumento. Es decir, un argumento es una pieza de información que se pasa de la llamada de una función hacia la función misma.

#### 4.2.3. Funciones que devuelven un valor

Ahora veamos un ejemplo sencillo de una función que requiere de parámetros de entrada y que además devuelve un valor.

```
[8]: def suma(a, b):  
      """Calcula la suma de dos números"""  
      c = a + b  
      return c  
  
      resultado = suma(3, 5)  
      print(resultado)
```

Este ejemplo muestra una nueva propiedad sobre las funciones. Hemos utilizado la palabra **return** para ser capaces de almacenar el resultado de la suma en una variable al momento de invocar la función. En realidad, la variable «c» definida dentro de la función no es necesaria. Hay una forma más simple de escribir una función que haga la misma tarea:

```
[9]: def sumar(a, b):
      """Calcula la suma de dos números"""
      return a + b
```

La función «suma» y la función «sumar» son totalmente equivalentes. Al momento de declarar funciones, intenta escribirlas de la manera más simple posible.

Veamos un ejemplo más. La siguiente función toma un número como parámetro de entrada y devuelve **True** si el número es par, o **False** si es impar.

```
[10]: def es_par(n):
       """Muestra si es par o no"""
       return n % 2 == 0
```

Comprobamos si los números 7 y 10 son pares o no:

```
[11]: es_par(7)
```

```
[11]: False
```

```
[12]: es_par(10)
```

```
[12]: True
```

### 4.3. Más sobre funciones con parámetros

Como habrás notado, es posible definir funciones con más de un parámetro. En la práctica, una función puede admitir una cantidad arbitraria de argumentos. Debido a esto, existen diferentes tipos de argumentos que pueden utilizarse en las funciones. A continuación revisaremos únicamente dos de ellos y cómo podemos combinarlos.

#### 4.3.1. Argumentos con un valor predeterminado

Al momento de declarar una función es posible especificar un valor predeterminado para uno o más de sus argumentos. Esto resulta bastante útil, ya que permite invocar a la función con menos argumentos de los definidos. Por ejemplo:

```
[13]: def saludar(name, repeat=True, bye='¡Adiós!'):
      """Muestra un saludo y despedida personalizados"""

      print(f'Hola, {name}.')
      if repeat:
          print(f'Hola de nuevo, {name}.')
      print(bye)
```

Esta función se compone de un argumento obligatorio llamado «name» y dos argumentos opcionales, llamados «repeat» y «bye». El argumento name es obligatorio porque no tiene ningún valor asignado por defecto y sin él, la función devolverá un error.

De manera concreta, la función saludar() puede ser invocada de muchas formas. La primera, brindando solo el argumento obligatorio:

```
[14]: saludar('Marty McFly')

Hola, Marty McFly.
Hola de nuevo, Marty McFly.
¡Adiós!
```

Ya que se ingresó un solo valor, la función le asignó ese valor al primer parámetro definido en la función. En este caso, al parámetro name. A este tipo de asignaciones se les llama argumentos posicionales, porque se asignan según el orden en que se proporcionan.

La segunda forma de invocar la función es brindando uno de los argumentos opcionales:

```
[15]: saludar('Marty McFly', False)

Hola, Marty McFly.
¡Adiós!
```

O como tercera opción, brindando todos los argumentos:

```
[16]: saludar('Marty McFly', False, 'Nos vemos')

Hola, Marty McFly.
Nos vemos
```

### 4.3.2. Argumentos de palabras clave

Las funciones también pueden ser invocadas haciendo uso de los argumentos de palabras clave (en inglés llamados *keyword arguments*), los cuales tienen la forma «kwarg=value». En esencia, al invocar la función se especifica el nombre del argumento y se le asigna un valor. Para comprenderlo, revisemos de nuevo la función `saludar()`. Esta función puede ser invocada en cualquiera de las siguientes formas:

```
[17]: saludar('Dr. Brown')           # 1 argumento posicional
```

```
Hola, Dr. Brown.  
Hola de nuevo, Dr. Brown.  
¡Adiós!
```

```
[18]: saludar(name='Dr. Brown')     # 1 argumento de palabra clave
```

```
Hola, Dr. Brown.  
Hola de nuevo, Dr. Brown.  
¡Adiós!
```

```
[19]: saludar(name='Dr. Brown', repeat=False) # 2 argumentos palabra clave
```

```
Hola, Dr. Brown.  
¡Adiós!
```

```
[20]: saludar(repeat=False, name='Dr. Brown') # 2 argumentos palabra clave
```

```
Hola, Dr. Brown.  
¡Adiós!
```

```
[21]: saludar('Dr. Brown', False, 'Nos vemos') # 3 argumentos posicionales
```

```
Hola, Dr. Brown.  
Nos vemos
```

```
[22]: saludar('Dr. Brown', bye='Nos vemos') # 1 posicional, 1 palabra clave
```

```
Hola, Dr. Brown.  
Hola de nuevo, Dr. Brown.  
Nos vemos
```

Pero todas las siguientes formas de invocarla son inválidas y resultan en un error:

```
[23]: saludar()
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[25], line 1
----> 1 saludar()

TypeError: saludar() missing 1 required positional argument: 'name'
```

El mensaje de error en este caso es bastante claro: se invocó a la función sin brindarle el argumento obligatorio. Intenta lo siguiente:

```
[24]: saludar(name='Dr. Brown', False)
```

```
Cell In[26], line 1
    saludar(name='Dr. Brown', False)
                                ^
SyntaxError: positional argument follows keyword argument
```

El error ocurre porque se intenta usar un argumento posicional después de un argumento de palabra clave y eso no está permitido en Python. Nuevamente, intenta:

```
[25]: saludar(False, name='Dr. Brown')
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[27], line 1
----> 1 saludar(False, name='Dr. Brown')

TypeError: saludar() got multiple values for argument 'name'
```

Este error se debe a que Python asigna el valor `False` al argumento `name` ya que ese valor se ingresa al principio como argumento posicional. Sin embargo, luego se ingresa el valor `'Dr. Brown'` al argumento `name` como argumento de palabra clave. De modo que se le está intentando asignar más de un valor al mismo argumento y eso resulta en un error. Para terminar, intenta ejecutar lo siguiente:

```
[26]: saludar(action='Repetir')
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[28], line 1
----> 1 saludar(action='Repetir')

TypeError: saludar() got an unexpected keyword argument 'action'
```

En este caso, el error se debe a que se intenta invocar a la función con un argumento que no está definido.

En conclusión, puedes observar lo siguiente al momento de invocar una función:

- Los argumentos de palabra clave deben ir después de los argumentos posicionales.
- Todos los argumentos de palabra clave deben coincidir con uno de los argumentos aceptados por la función (por ejemplo, `action` no es un argumento válido para la función `saludar`) y el orden en el que aparecen no es importante. Esto también incluye a los argumentos no opcionales (por ejemplo, `saludar(name='Dr. Brown')` es una forma válida de invocar la función).
- Ningún argumento puede recibir más de un solo valor.

#### 4.4. Problemas

Los siguientes problemas están pensados para que te familiarices aún más con Python y las funciones. El objetivo es que realices las tareas que se te piden SIN utilizar las funciones predefinidas de Python.

1. Escribe una función que encuentre el mayor entre dos números.
2. Escribe una función que encuentre el mayor entre tres números.
3. Escribe una función que encuentre el mayor número en una lista.
4. Escribe una función que calcule la suma de los elementos en una lista.
5. Escribe una función que calcule el producto de los elementos en una lista.
6. Escribe una función que calcule el factorial de un número.
7. Escribe una función que invierta los caracteres de una variable de tipo `str`. Es decir, si se ingresa la palabra `'AMOR'`, la función debe devolver la palabra `'ROMA'`.
8. Escribe una función que compruebe si una palabra/oración es un palíndromo.
9. Escribe una función que calcule la cantidad de caracteres que tiene una palabra.
10. Escribe una función que calcule la cantidad de caracteres que tienen todas las palabras en una lista.