

Reducción de datos CCD con IRAF y Python

Alexis Andrés

23 de agosto de 2024

Índice general

Clase 1 Introducción a Python	2
1 Nociones básicas	2
1.1 Python en diferentes sistemas operativos	2
1.1.1 Python en Windows	3
1.1.2 Python en Linux	3
1.1.3 Python en Mac OSX	4
1.2 Ejecutando Python	5
1.2.1 Creando un entorno virtual	6
1.2.2 El editor Geany	7
1.2.3 Jupyter Notebook	9
1.2.4 Visual Studio Code y PyCharm	11
1.3 Python como calculadora	11
1.3.1 Operaciones aritméticas	11
1.3.2 Operaciones matemáticas avanzadas	13
Clase 2 Variables y contenedores	15
2 Introducción	15
2.1 Variables	15
2.1.1 Variables tipo int/float	16
2.1.2 Variables de tipo string	16
2.1.3 Comentarios	17
2.1.4 Variables tipo bool	18
2.2 Contenedores	19
2.2.1 Listas	19
2.2.2 Tuplas	22
2.2.3 Conjuntos	23
2.2.4 Diccionarios	24

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.

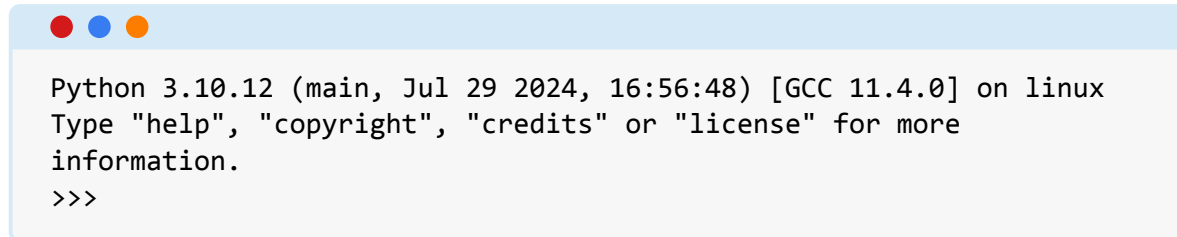
Al igual que con los sistemas Linux, es posible que el resultado sea un error o un mensaje con algo similar a **2.x.x**, lo que significa que la versión instalada es Python 2. Si cualquiera de las dos opciones anteriores es cierta, intenta escribiendo en la terminal **python3 --version** y presiona Enter.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

A terminal window with a light blue header bar containing three colored circles (red, blue, orange). The terminal text shows the Python version and system information, followed by the Python prompt.

```
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:

A terminal window with a light blue header bar containing three colored circles (red, blue, orange). The terminal text shows a print statement being executed and its output.

```
>>> 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).

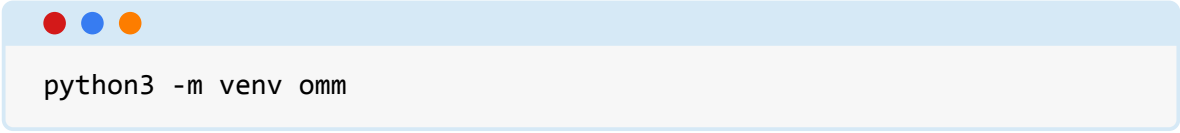
1.2.1. Creando un entorno virtual

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:

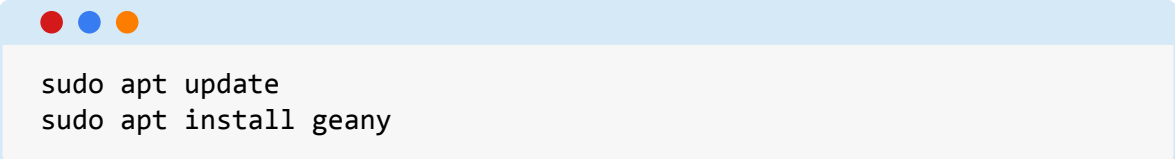
```
C:\Users\user\omm\Scripts\python "%f"
```

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.

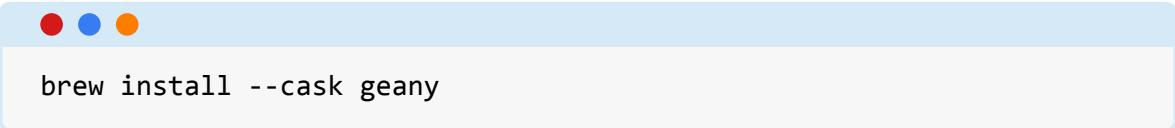
Geany en Linux y Mac OSX: Para instalar Geany en Linux, es suficiente con que ejecutes los siguientes comandos en una terminal:

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

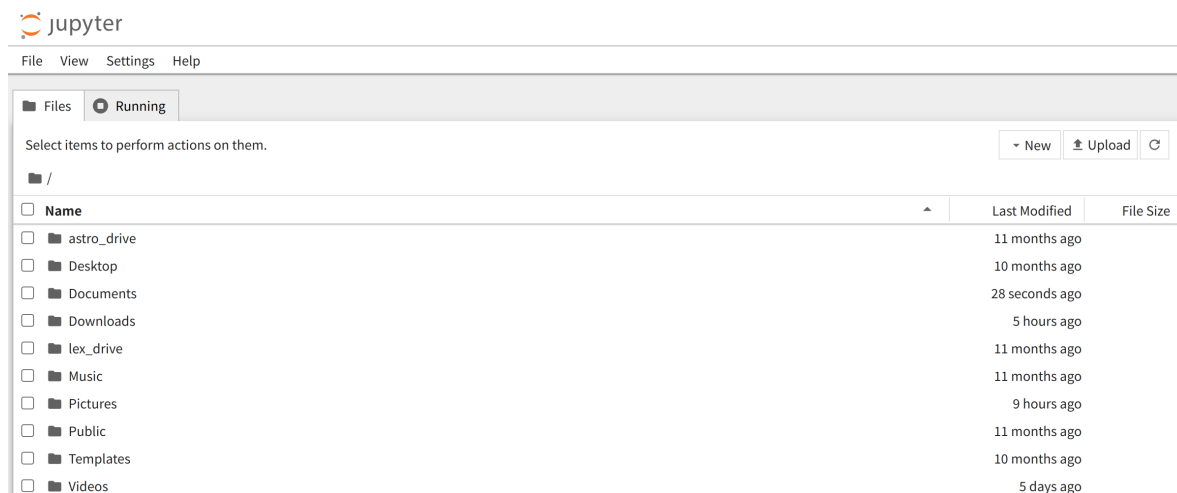


Figura 1.1: Interfaz gráfica de Jupyter Notebook

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

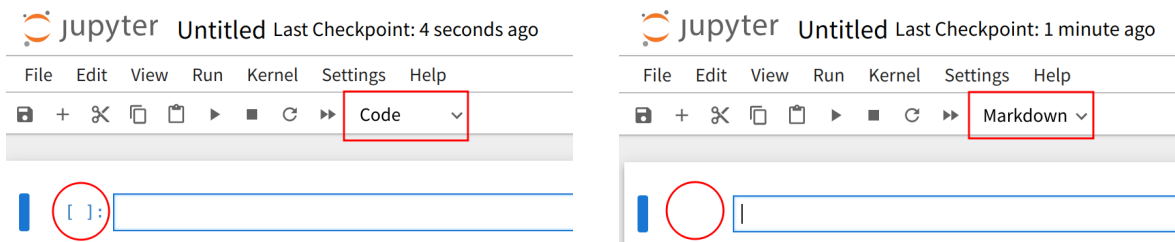


Figura 1.2: Opciones disponibles para escribir en Jupyter

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 4×10^3 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 4×10^3 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>></code>	Mayor que
<code><</code>	Menor que
<code>>=</code>	Mayor o igual que
<code><=</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))])
```