



Fundamentos de programación en Python



I. Introducción

II. Objetivos

III. El lenguaje Python y el entorno Jupyter Notebook

IV. Elementos básicos de Python

V. Estructuras de control

VI. Estructuras de datos

VII. Funciones

VIII. Excepciones

IX. Importación de módulos

X. Gestión de archivos

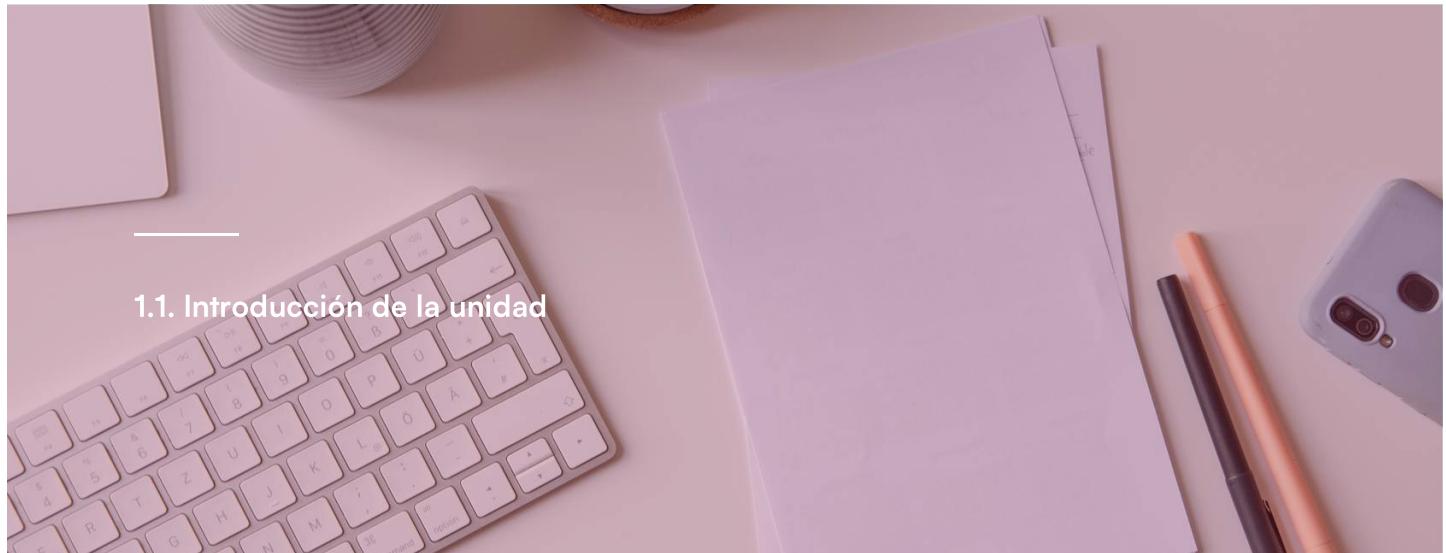
XI. Resumen

XII. Caso práctico con solución

XIII. Glosario

XIV. Bibliografía

I. Introducción



1.1. Introducción de la unidad

Un elemento básico para cualquier científico de datos es el **conocimiento de algún lenguaje de programación que le permita desarrollar programas con los que procesar la información**. Aunque sería posible utilizar cualquier lenguaje de propósito general, hay algunos que incluyen funcionalidades específicas para las operaciones más comunes en este ámbito. En este sentido, existen varias alternativas.

Actualmente, hay dos lenguajes de programación que cubren las necesidades de procesamiento para un proyecto de análisis de datos: el lenguaje R y el lenguaje Python.

En esta unidad, se va a estudiar el lenguaje Python. Se trata de un lenguaje de propósito general al que se han añadido, en forma de librerías, las funcionalidades necesarias para análisis de datos de cualquier complejidad. Presenta funciones y estructuras de datos eficientes semejantes al lenguaje R. Asimismo, este lenguaje se ha popularizado por varios motivos, entre los que destacan su **facilidad de aprendizaje y su uso intensivo en la industria para este tipo de tareas**.

A parte de la facilidad de aprendizaje que ofrece el lenguaje, hay otros motivos por los que es muy popular:

- **Dispone de una amplia comunidad de desarrolladores.** Continuamente se están desarrollando nuevas funcionalidades en forma de librerías, de manera que es fácil encontrar en sus repositorios la funcionalidad que permita ahorrar tiempo en los desarrollos.
- Al tratarse de un lenguaje muy popular, **contratar personal cualificado en Python es más sencillo** que para otras disciplinas.

- Al tratarse de un lenguaje de propósito general, **se puede utilizar para construir cualquier tipo de aplicación**, no solo orientadas a datos.
- Ofrece una **gran capacidad de abstracción sobre el sistema, con un lenguaje muy expresivo**, de manera que se puede hacer uso de todo el hardware de la máquina, reduciendo al mínimo la cantidad de código necesaria.

CONTINUAR

En esta unidad se estudiará, en primer lugar, uno de los entornos de desarrollo más utilizados por analistas y científicos de datos, denominado **Jupyter Notebook**, y, a continuación, se revisarán los **principales elementos del lenguaje de programación**.

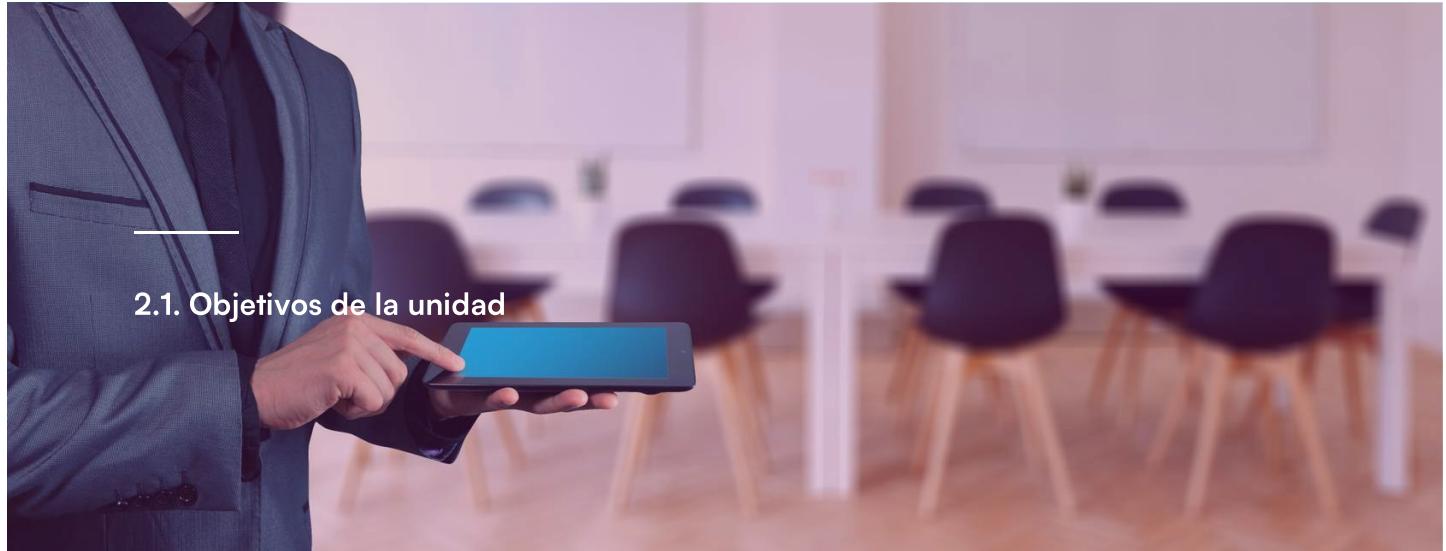
Dentro de estos elementos del lenguaje, se profundizará en cuatro pilares básicos:

- Sintaxis del lenguaje. Es preciso conocer los bloques más básicos que permitan construir los algoritmos.
- Estructuras de control que permitan ejecutar sentencias del programa en función de determinadas condiciones.
- Estructuras de datos que permitan manipular información compleja.
- Funciones que permitan reutilizar el código.

Ten en cuenta:

En esta unidad se incluyen ejercicios que muestran la estructura y los elementos de Python. Se recomienda ejecutarlos en cuadernos de Jupyter Notebook, ya que será beneficioso para el entendimiento de la sintaxis y relevancia del lenguaje. Al finalizar la unidad, el alumno será capaz de leer y codificar programas básicos en Python.

II. Objetivos



2.1. Objetivos de la unidad

Los objetivos para el estudio de esta unidad son:

- 1 Conocer el entorno Jupyter Notebook para el desarrollo de programas en Python.
- 2 Conocer los elementos básicos del lenguaje Python.
- 3 Conocer las estructuras de control y las principales estructuras de datos del lenguaje Python.
- 4 Saber desarrollar programas de complejidad media y simple con el lenguaje Python.
- 5 Entender lo que hace una parte de código Python de complejidad media y simple.
- 6 Resolver problemas de distinta índole.

III. El lenguaje Python y el entorno Jupyter Notebook

3.1. El lenguaje de programación Python

Python es un **lenguaje de programación de alto nivel creado por Guido van Rossum**. Se desarrolla como un proyecto de **código libre**, de manera que existe una comunidad de desarrolladores que mantienen el lenguaje, gestionan las distintas versiones y crean librerías para aumentar su funcionalidad.

Algunas de sus características son:

Es un lenguaje interpretado



Por lo que no es necesario compilar el código antes de su ejecución.

Es multiparadigma



En este sentido, permite varios estilos de programación: imperativo, orientado a objetos y funcional.

Es multiplataforma



Python es un lenguaje disponible en los principales sistemas operativos (Windows, Linux y Mac).

Posee un tipado dinámico



El tipo de los datos es inferido en tiempo de ejecución, de manera que no es necesario declarar el tipo de sus variables y permite conversiones dinámicas de los tipos de los datos.

En comparación con otros lenguajes de programación, **Python es un lenguaje simple, fácil de leer y escribir y simple de depurar**. Por estas razones es fácil de aprender, de manera que la curva de aprendizaje es corta.

Asimismo, Python **cuenta con una gran cantidad de librerías, tipos de datos y funciones incorporadas en el propio lenguaje**, lo que lo dota de una gran capacidad de procesamiento. En particular, se utiliza ampliamente en el ámbito del análisis de datos y, en general, en tareas de procesamiento de ciencias e ingeniería.

Desde el punto de vista del análisis de datos, dispone de una amplia variedad de librerías y herramientas como **Numpy, Pandas, Matplotlib, Scipy, Scikit-learn, Theano, TensorFlow**, etc.¹

¹[Popular Math libraries in Python](#).

CONTINUAR

3.2. El entorno Jupyter Notebook

En cualquier lenguaje de programación, las herramientas de edición constituyen un elemento esencial. En general, este tipo de herramientas **implementan funcionalidades como el autocompletado de palabras del lenguaje de programación, ayuda interactiva, coloreado de las estructuras sintácticas del lenguaje, depuración de errores, compilación o interpretación y ejecución del programa**.

Jupyter lleva estas funcionalidades a un espacio más visual, ya que permite **no solo desarrollar código Python, sino también integrar fragmentos de código (ejecutable) con otros contenidos multimedia o textuales**. Esto convierte a Jupyter en una potente herramienta de documentación y elaboración de informes analíticos.²

²[Ejemplo de Jupyter Notebook](#).

Los documentos generados se visualizan con un navegador (Explorer, Chrome, Firefox...) y **pueden incluir cualquier elemento accesible a una página web**, además de permitir la ejecución de código escrito en el lenguaje de programación Python.

Jupyter **contiene todas las herramientas científicas estándar de Python que permiten realizar tareas propias** en el contexto del análisis de datos: **importación y exportación, manipulación y transformación, visualización, etc.**

Para instalarlo, lo mejor es hacerlo a través de la *suite* de Anaconda, la cual **incluye un conjunto de herramientas de desarrollo para Python entre las que se encuentra el propio Jupyter Notebook**. Anaconda se puede descargar desde la página web de Anaconda.³ En la zona de descargas, aparecen dos versiones. Se debe bajar la versión para Python 3.x (figura 1).

Anaconda Installers

Windows 	MacOS 	Linux 
Python 3.8	Python 3.8	Python 3.8
64-Bit Graphical Installer (477 MB)	64-Bit Graphical Installer (440 MB)	64-Bit (x86) Installer (544 MB)
32-Bit Graphical Installer (409 MB)	64-Bit Command Line Installer (433 MB)	64-Bit (Power8 and Power9) Installer (285 MB)
		64-Bit (AWS Graviton2 / ARM64) Installer (413 M)
		64-bit (Linux on IBM Z & LinuxONE) Installer (292 M)

Figura 1. Zona de descargas de Anaconda.

Fuente: elaboración propia.

³[Página web de Anaconda](#).

CONTINUAR

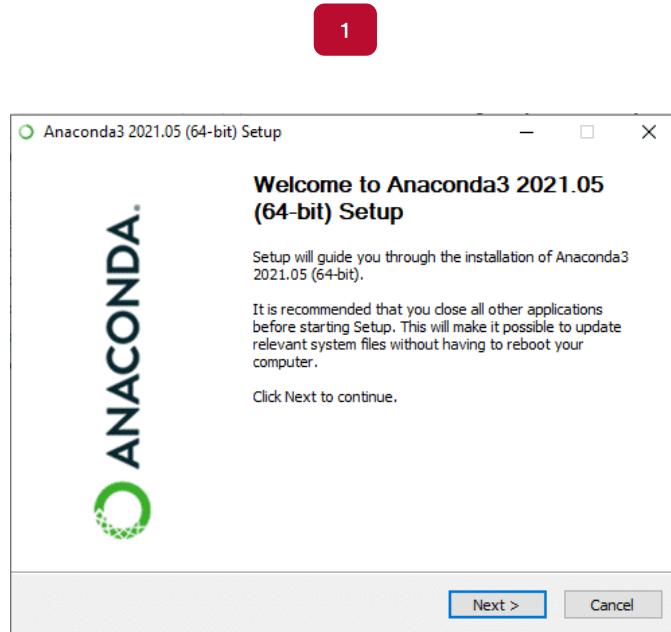


Figura 2. Asistente de instalación de Anaconda.

Fuente: elaboración propia.

Una vez descargado el software, se pueden seguir las instrucciones de instalación para cada sistema operativo que aparecen en la misma página de descargas.⁴

No obstante, a continuación se sigue el procedimiento de instalación estándar de la versión 64-bit para Windows.

⁴[Página web de Anaconda. Installation.](#)

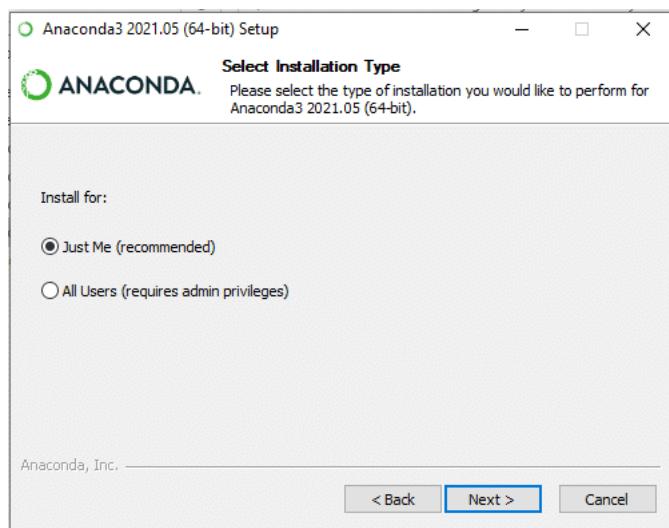


Figura 3. Ámbito de la instalación.

Fuente: elaboración propia.

Se selecciona el ámbito de uso de la *suite* de Anaconda (local al usuario, o global al sistema). Para este ejemplo, se sigue la opción recomendada (figura 3).

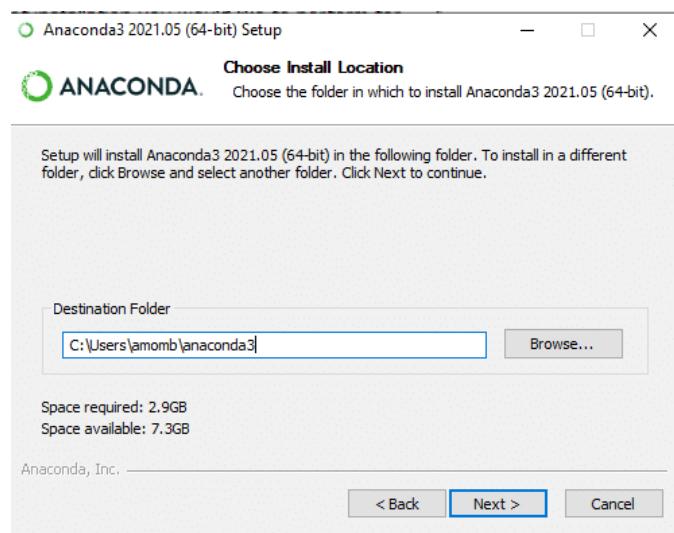


Figura 4. Ubicación de la instalación.

Fuente: elaboración propia.

Se selecciona la ruta donde se instalará Anaconda y se pulsa *Next* (figura 4).

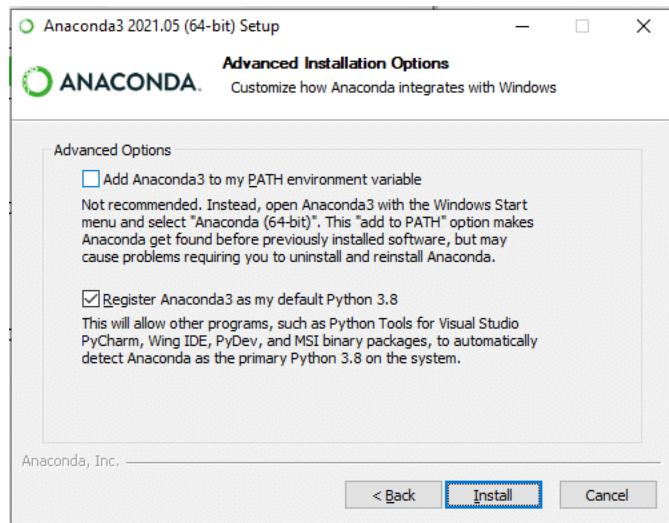


Figura 5. Opciones avanzadas.

Fuente: elaboración propia.

Se puede añadir Anaconda al **PATH** del sistema. Esto implica que los comandos de Anaconda (Python, etc.) se podrán ejecutar desde cualquier parte del sistema. Sin embargo, esta opción no es recomendable, puesto que puede causar problemas en caso de desinstalar y volver a instalar Anaconda. Asimismo, se registra la versión Python de Anaconda como el intérprete Python por defecto del sistema (figura 5).

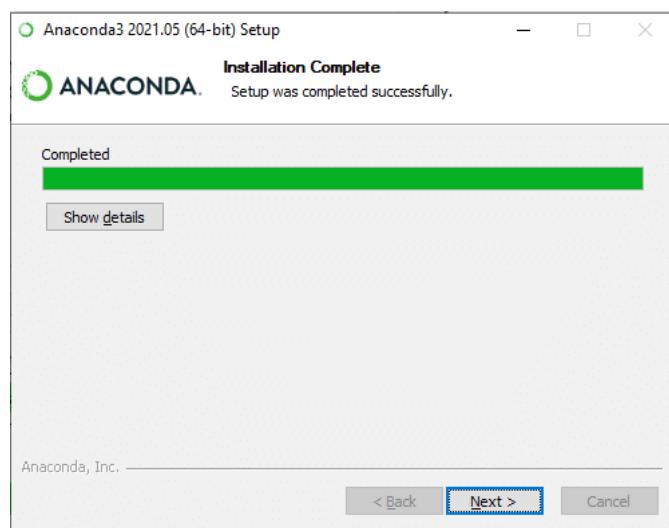


Figura 6. Instalación completada.

Fuente: elaboración propia.

En este punto se inicia el proceso de instalación, que puede durar algunos minutos. Una vez finalizado, se pulsa en *Next* (figura 6).

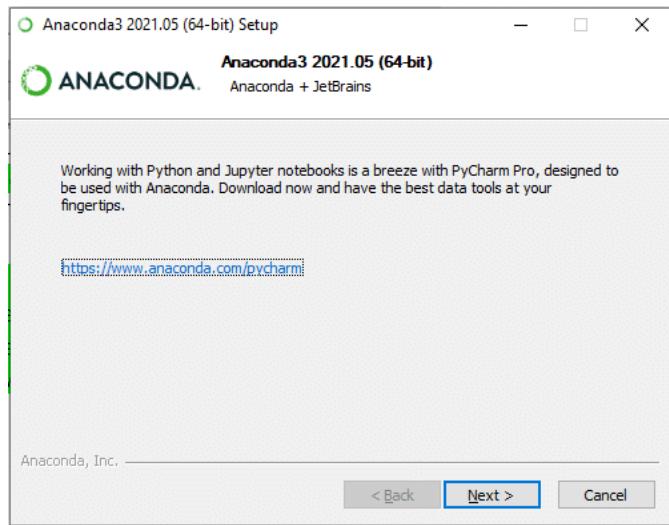


Figura 7. Anaconda + PyCharm.

Fuente: elaboración propia.

El propio asistente da la recomendación de usar PyCharm⁵ como entorno integrado de desarrollo (figura 7). Instalarlo es opcional y no es necesario para el estudio de esta unidad. No obstante, se recomienda su instalación, ya que PyCharm ofrece herramientas adicionales que ayudan en la gestión de grandes proyectos (navegación entre funciones, refactorización de código, etc.).

⁵[PyCharm Download](https://www.anaconda.com/pycharm).

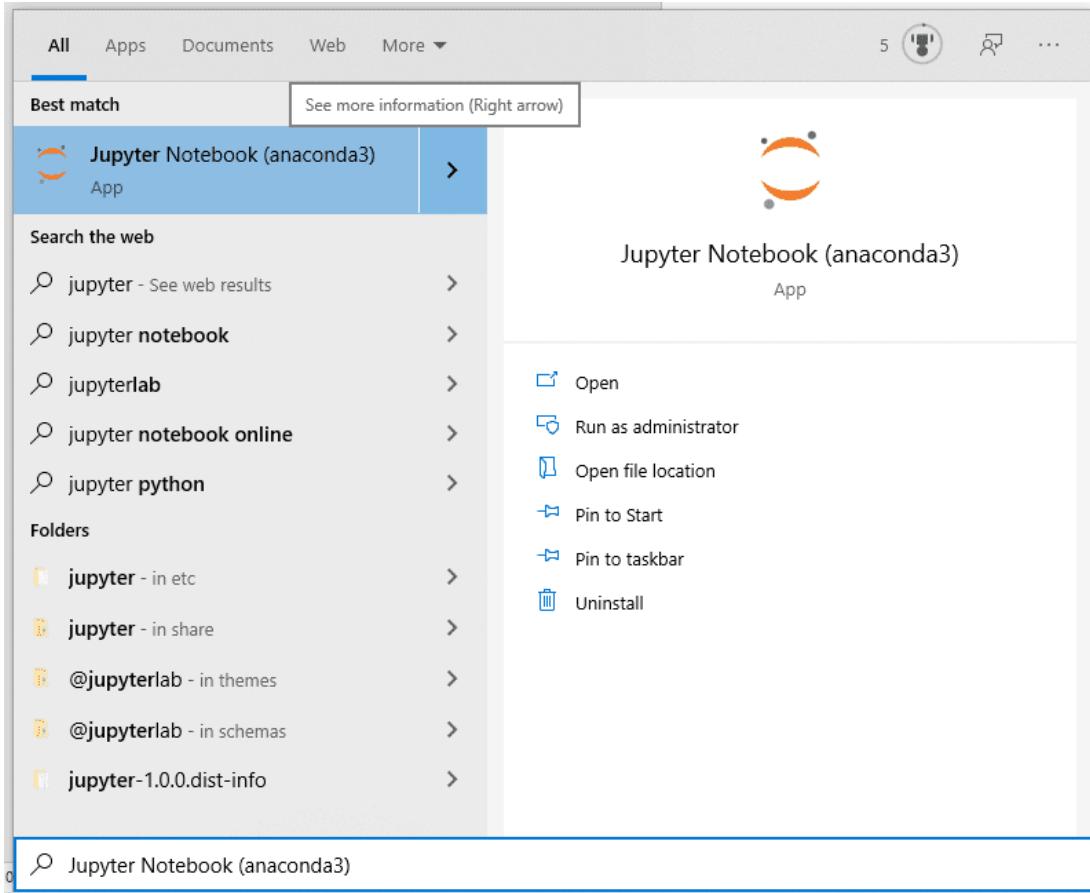
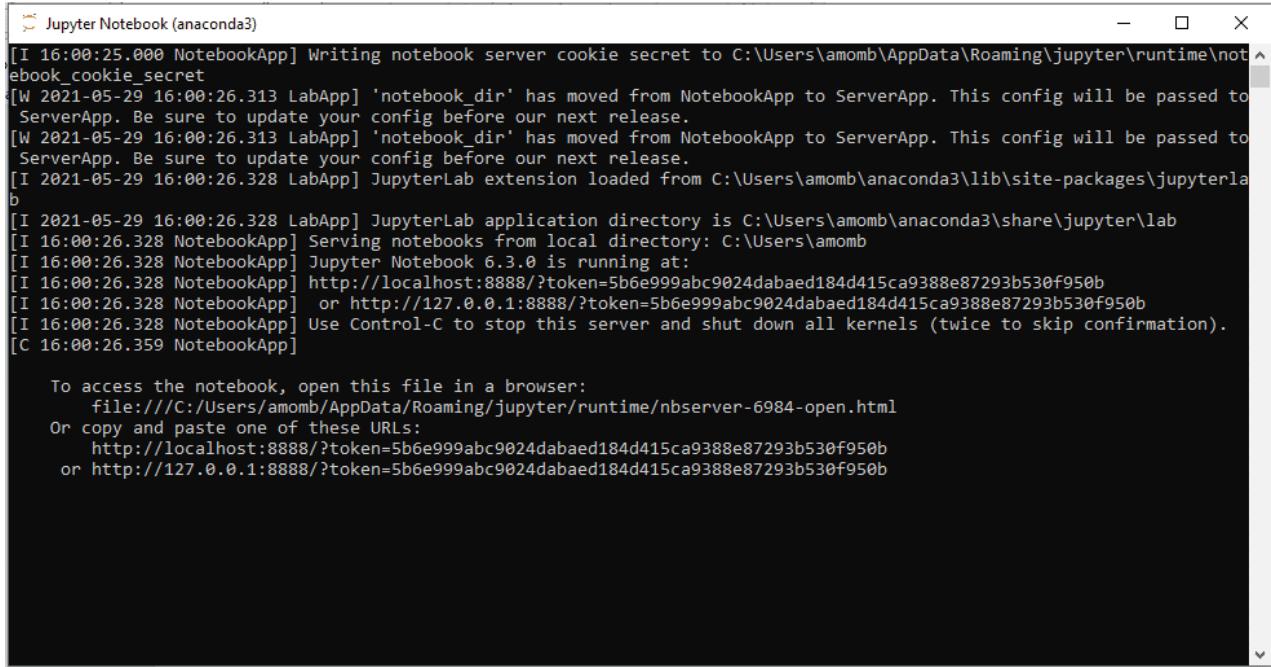


Figura 8. Ejecución de Jupyter Notebook.

Fuente: elaboración propia.

Para ejecutar Jupyter Notebook, basta con buscarlo en la lista de aplicaciones instaladas y pulsar en *Open* (figura 8).

Jupyter Notebook es una aplicación web bajo el modelo cliente-servidor. Al ejecutar Jupyter, se iniciará un servicio que se ejecuta localmente, por defecto, en la URL <http://127.0.0.1:8888>, y que es accesible desde el navegador, por lo cual aparece un nuevo terminal donde se ejecuta la herramienta (figura 9). Este terminal no debe cerrarse mientras se esté trabajando con Jupyter Notebook, de lo contrario el servicio se detendrá.



The screenshot shows a terminal window titled "Jupyter Notebook (anaconda3)". The window displays log messages from the Jupyter Notebook application. Key messages include:

- [I 16:00:25.000 NotebookApp] Writing notebook server cookie secret to C:\Users\amomb\AppData\Roaming\jupyter\runtime\notebook_cookie_secret
- [W 2021-05-29 16:00:26.313 LabApp] 'notebook_dir' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to update your config before our next release.
- [W 2021-05-29 16:00:26.313 LabApp] 'notebook_dir' has moved from NotebookApp to ServerApp. This config will be passed to ServerApp. Be sure to update your config before our next release.
- [I 2021-05-29 16:00:26.328 LabApp] JupyterLab extension loaded from C:\Users\amomb\anaconda3\lib\site-packages\jupyterlab
- [I 2021-05-29 16:00:26.328 LabApp] JupyterLab application directory is C:\Users\amomb\anaconda3\share\jupyter\lab
- [I 16:00:26.328 NotebookApp] Serving notebooks from local directory: C:\Users\amomb
- [I 16:00:26.328 NotebookApp] Jupyter Notebook 6.3.0 is running at:
- [I 16:00:26.328 NotebookApp] http://localhost:8888/?token=5b6e999abc9024dabaed184d415ca9388e87293b530f950b
- [I 16:00:26.328 NotebookApp] or http://127.0.0.1:8888/?token=5b6e999abc9024dabaed184d415ca9388e87293b530f950b
- [I 16:00:26.328 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
- [C 16:00:26.359 NotebookApp]

Below the log messages, there is a prompt for accessing the notebook:

To access the notebook, open this file in a browser:
file:///C:/Users/amomb/AppData/Roaming/jupyter/runtime/nbserver-6984-open.html
Or copy and paste one of these URLs:
<http://localhost:8888/?token=5b6e999abc9024dabaed184d415ca9388e87293b530f950b>
or <http://127.0.0.1:8888/?token=5b6e999abc9024dabaed184d415ca9388e87293b530f950b>

Figura 9. Terminal de ejecución de Jupyter Notebook.

Fuente: elaboración propia.

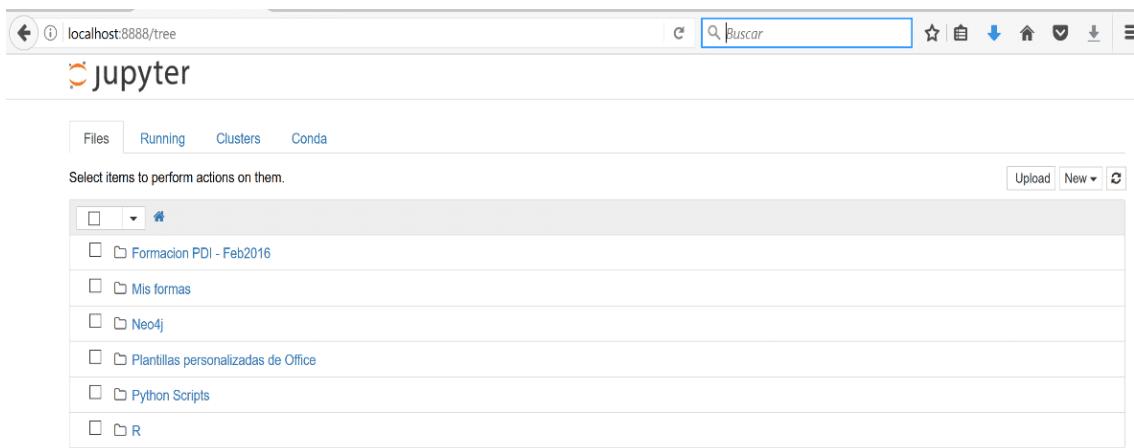


Figura 10. Interfaz principal de Jupyter Notebook.

Fuente: elaboración propia.

A la vez, se lanza un navegador donde se puede acceder a la interfaz principal de Jupyter Notebook (figura 10).

Esta interfaz actúa como un navegador de archivos y es posible moverse entre distintas carpetas —basta pinchar sobre la carpeta correspondiente y se accede a su contenido—, además de crear nuevos ficheros y carpetas. Para ello se pulsa sobre el desplegable que aparece en la parte derecha, denominado *New*, y allí se selecciona *Text File* o *Folder*, según lo que se quiera crear. También es posible renombrar y eliminar las carpetas y archivos. Para ello basta con seleccionar la correspondiente carpeta o archivo. Una vez seleccionado, en la parte superior aparecen las opciones *Rename* y el icono de la papelera, que permiten renombrar y eliminar respectivamente.

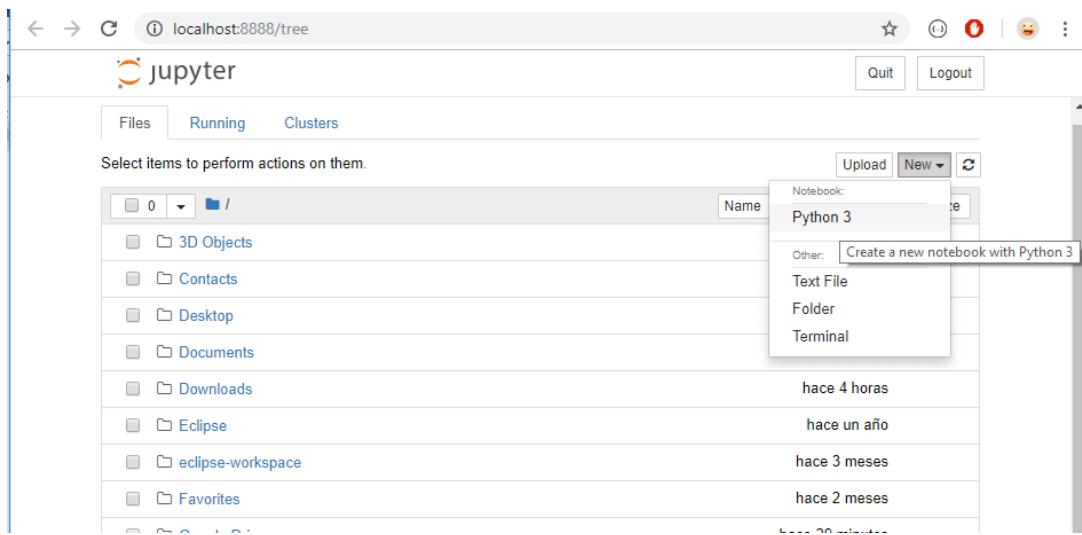


Figura 11. Creación de un notebook de Jupyter Notebook.

Fuente: elaboración propia.

Los notebooks de Jupyter son unos archivos con extensión .ipynb, en los que se puede escribir código Python ejecutable, texto, dibujar gráficas y otras operaciones más. Para crearse un nuevo notebook basta con pulsar sobre el desplegable que aparece en la parte superior derecha, denominado *New*, y seleccionar la opción *Python 3* que se muestran debajo de *Notebook* (figura 11).

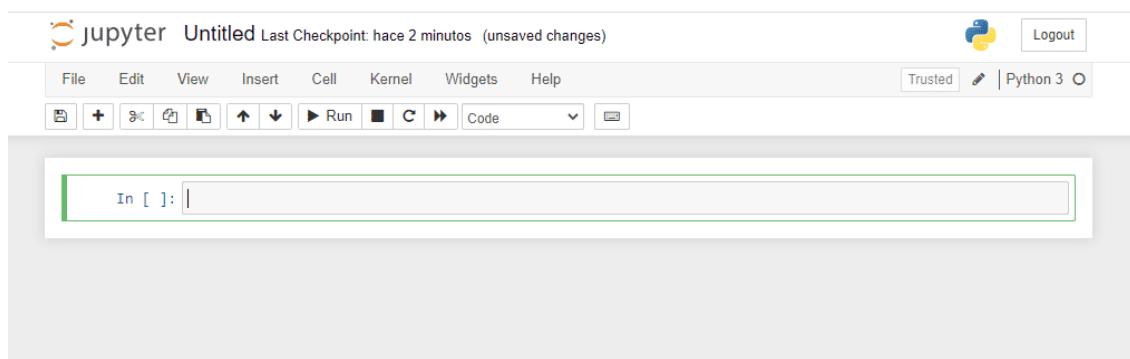


Figura 12. Notebook de Jupyter.

Fuente: elaboración propia.

Una vez pulsado, se carga el nuevo *notebook* creado (figura 12).



Figura 13. Elementos esenciales de un notebook.

Fuente: elaboración propia.

Los elementos esenciales de un notebook son (figura 13):

- **Título del notebook.** Cada notebook tiene un título asociado. Por defecto, aparece con el nombre "Untitled". Para modificarlo, basta con pulsar sobre "Untitled" y aparece una ventana en la que se puede modificar el nombre.
- **Menús y barra de herramientas.** En la parte superior de la interfaz del notebook, aparece un conjunto de menús con diferentes opciones para gestionar los archivos, para editar, visionar, etc.
- **Iconos de acceso rápido.** Iconos que permiten realizar las acciones más comunes sobre los elementos del notebook.
- **Celdas.** La unidad de edición de un notebook es la celda. Cada celda contiene código Python o la información que documenta dicho código. En este sentido, un notebook es una secuencia de celdas. Las celdas pueden ser de diferentes tipos según el contenido que almacenan:
 - **Markdown:** permite escribir texto formateado con el objetivo de documentar. Se usa el lenguaje de marcas Markdown.⁶
 - **Raw NBConvert:** son celdas que permiten escribir fragmentos de código sin que sean ejecutados.
 - **Heading:** permite embeder código html.
 - **Code:** sirven para escribir código Python ejecutable. Están marcadas por la palabra **In [n]** y están numeradas. El resultado de la ejecución de cada celda se muestra en el campo **Out [n]**, también numerado.

Para elegir el tipo de celda, se selecciona en un desplegable que aparece en la fila superior junto a los iconos.

⁶[Markdown Guide](#).

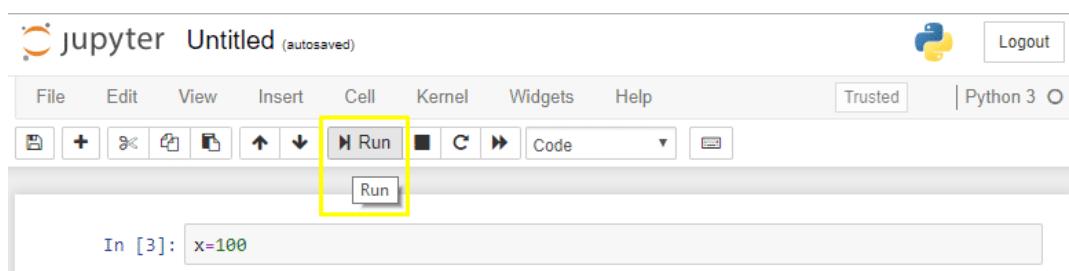


Figura 14. Ejecución de una celda.

Fuente: elaboración propia.

Todas las celdas son susceptibles de ejecutarse. La ejecución de una celda de código Python ejecutará el código escrito en la celda y producirá una salida. La ejecución de celdas de tipo Markdown dará formato al texto. Para ejecutar una celda, hay que colocarse en la celda y, posteriormente, pulsar el botón *Run* (figura 14). Es recomendable ver el video “Running Hello World on a Jupyter Notebook”⁷. En él se muestran los pasos para ejecutar su primer programa.

⁷Lumiwealth. “[Running Hello World on a Jupyter Notebook](#)”. YouTube, 18 de abril de 2020.

The screenshot shows a Jupyter Notebook interface. At the top is a menu bar with File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with various icons for file operations like Open, Save, and Run, along with a 'Code' dropdown. The main area contains two code cells. The first cell, labeled 'In [1]:', contains the Python code `def multiplica(a,b): return a*b`. The second cell, also labeled 'In [1]:', contains the command `multiplica(3,4)`. The output, labeled 'Out[1]:', is the result `12`.

Figura 15. Ejemplo de función.

Fuente: elaboración propia.

De igual forma, para interrumpir la ejecución se pulsa sobre el cuadrado. Para crear celdas nuevas, se pulsa sobre el botón con el icono del signo +.

El flujo normal de edición de una celda consiste en:

- Elegir el tipo de celda. Por defecto, son de tipo Code. Dependiendo del tipo de celda elegido, Jupyter lo interpretará de diferente manera.
- Una vez introducido el código o texto en la celda, se debe ejecutar. Para ello, se pulsa sobre el ícono en forma de flecha.
- A continuación, se genera una nueva celda para editar.
- El código definido en cada celda de tipo Code es visible al resto de celdas de este tipo. Es decir, si se define una variable o una función en una celda, se podrán utilizar en celdas posteriores.

Ejemplo

Por ejemplo, si se quiere crear una celda que contenga la función: `def multiplica(a,b): return a*b` y después invocarla con los valores 3 y 4, `multiplica(3,4)`, se haría como se muestra en la figura 15.

IV. Elementos básicos de Python

Variables:

Una variable es un **nombre que referencia un valor de cualquier tipo de dato**. Por ejemplo:

```
titulo="Cálculo de área de un círculo"  
pi=3.1416  
radio=5  
area=pi*(radio**2)
```

Una **sentencia de asignación** crea variables nuevas y las asocia a valores. Tiene la siguiente sintaxis:

nombre = valor

Por ejemplo:

```
mensaje = "Esto es un mensaje de prueba"  
n = 17  
pi = 3.1415926535897931
```

Para mostrar el valor de una variable, se puede usar la función *print(nombre_variable)*. Asimismo, se le puede pasar una lista de variables, separadas por *" "*, que serán impresas en orden, en la misma línea. Por ejemplo:

```
mensaje = "Esto es un mensaje de prueba"  
n = 17  
pi = 3.1415926535897931  
print(n)  
print(pi)
```

```

mensaje="Esto es un mensaje de prueba"
n=17
pi=3.1415926535897931
print(n)
17
print(pi)
3.141592653589793

```

Figura 16. Ejecución y salida del código.

Fuente: elaboración propia.

Las variables son de un tipo que coincide con el tipo del valor que referencian. El método `type()` indica el tipo de una variable. Por ejemplo:

```

type(mensaje)
type(n)

```

```
type (mensaje) #str
```

str

```
type (n) #int
```

int

Figura 17. Ejecución y salida del código.

Fuente: elaboración propia.

CONTINUAR

Python ofrece **cinco tipos de datos “primitivos”⁸**, denominados así porque constituyen la base para componer tipos de datos más complejos. Son:

INT	FLOAT	BOOL	STR
-----	-------	------	-----

INT

FLOAT

BOOL

STR

Enteros.

INT

FLOAT

BOOL

STR

Números reales.

INT

FLOAT

BOOL

STR

Valores booleanos, **True** y **False**.

INT

FLOAT

BOOL

STR

Cadenas de texto.

INT

FLOAT

BOOL

STR

Es un tipo de dato especial, que se corresponde con la ausencia de valor o valor nulo.

⁸[Understanding Python3 primitive data types.](#)

Atención:

Es importante destacar que hay una serie de restricciones a la hora de establecer nombres de variables⁹:

- Pueden ser arbitrariamente largos.
- Pueden contener tanto letras como números, así como el carácter “_”.
- Deben empezar con letras o el carácter “_”.
- No pueden ser palabras reservadas de Python.
- Los nombres de variables son sensibles a mayúsculas. Por tanto, **Nombre** y **nombre** serán dos variables diferentes.

⁹[Python Variable Names](#).

CONTINUAR

Antes de poder utilizar una variable, esta **debe existir** (porque se le ha asignado un valor en algún momento anterior en el código).

En el siguiente ejemplo **se muestra cómo se inicia una variable con el valor 0 y posteriormente se utiliza su valor en una segunda operación aritmética: a su valor se le suma 1 y el resultado se vuelve a guardar en la misma variable**. Es decir, se puede utilizar una misma variable en ambos lados de una operación de asignación (=). En el lado derecho se lee su valor actual y en el izquierdo se actualiza con el valor resultante de la operación):

```
x = 0  
x = x + 1
```

Por ejemplo:

- Programa que suma dos números e imprime el resultado:

```
n1 = 1  
n2 = 3  
suma = n1 + n1  
print(n1, " + ", n2, " = ", suma)
```

- Programa que calcula el área de un cuadrado:

```
lado = 3  
area = lado * lado
```

```
print ("El área de un cuadrado que mide ",lado, "cm por lado es ",area)
```

CONTINUAR

4.1. Palabras reservadas

Python reserva 31 palabras clave para su propio uso:

```
import keyword  
print(keyword.kwlist)
```

```
import keyword  
print(keyword.kwlist)  
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del',  
'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'la  
mbda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Figura 18. Ejecución y salida del código.
Fuente: elaboración propia.

CONTINUAR

4.2. Operadores

Los operadores **son símbolos especiales que representan cálculos, como la suma o la multiplicación**. Los valores a los cuales se aplican esos operadores reciben el nombre de **operando**.

El resultado de cada operación dependerá del tipo de dato de cada operando. Incluso se pueden mezclar tipos de datos diferentes, de manera que el tipo de dato del resultado de la operación será el mismo que el del operando menos significativo, es decir, el más genérico.

Ejemplo

La suma de dos variables de tipo **int** producirá como resultado otro **int**. Sin embargo, la suma de un **int** y un **float** será de tipo **float**.

- $i+j$, suma.
- $i-j$, resta.
- $i*j$, multiplicación.
- i/j , división de dos números. Si son enteros, el resultado es un entero, y si son reales, el resultado es un real.
- $i//j$, cociente de la división entera.
- $i\%j$, resto de la división entera (también denominado operador módulo).
- $i**j$, que representa i elevado a la potencia j .
- $i==j$, operador booleano que devuelve True si i es igual a j y False en caso contrario.
- $i!=j$, operador booleano que devuelve True si i es distinto de j y False en caso contrario.
- $i>j$, operador booleano para comprobar si i es mayor que j , y de forma similar: \geq , $<$, \leq , respectivamente mayor o igual, menor y menor o igual.

Ejemplo

- $x > 0 \text{ and } x < 10$ es verdadero solo cuando x es mayor que 0 y menor que 10.
- $n \% 2 == 0 \text{ or } n \% 3 == 0$ es verdadero si el número es divisible por 2 o por 3.
- $\text{not } (x > y)$ es verdadero si x es menor o igual que y .



Hay que tener en cuenta que cualquier número distinto de cero se interpreta como “verdadero”. Por ejemplo: 23 and True

23 and True

True

Figura 22. Ejecución y salida del código.
Fuente: elaboración propia.

Algunos ejemplos de uso de estos operadores booleanos son:

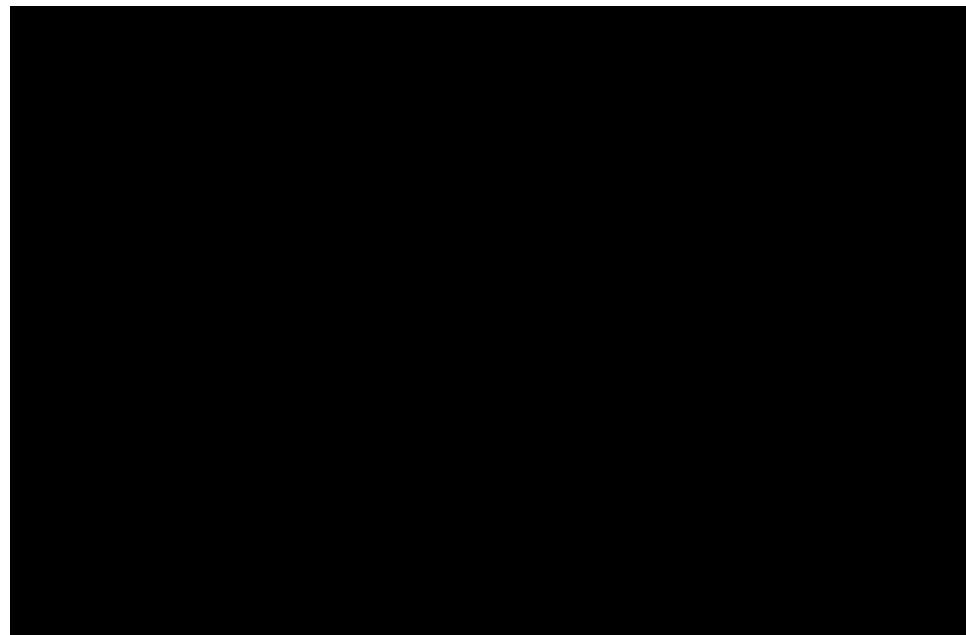
```
7 == 7  #True
7 != 7  #False
7 > 7  #False
7 >= 7  #True
10 > 1  #True
10 < 1  #False
10 <= 1  #False
1 < 10  #True
```

OPERADORES BOOLEANOS

OPERADORES LÓGICOS

Algunos ejemplos de uso de estos operadores lógicos son:

```
8>5 and 8>6  #True
8>5 and 8>9  #False
8>5 or 8>6  #True
8>5 or 8>9  #True
8>11 or 8>9  #False
8>11  #False
not(8>11)  #True
8>5  #True
not(8>5)  #False
```



V. Estructuras de control

Una vez vistas las sintaxis que ofrece Python para componer expresiones, es momento de profundizar en el desarrollo de algoritmos más complejos.

Para componer estos algoritmos, se utilizan **estructuras o sentencias de control** que permiten ejecutar conjuntos de expresiones en función de condiciones.

Para agrupar estos conjuntos de expresiones asociadas a una estructura de control, **Python obliga a utilizar diferentes niveles de sangrado del código**, de manera que esos bloques se puedan diferenciar de manera muy visual. La profundidad y los caracteres utilizados para definir este sangrado no son relevantes en Python, aunque sí se ha definido un estándar dentro de la norma PEP8 (norma que define un conjunto de buenas prácticas para la programación en Python), que establece que los sangrados deben ser de cuatro espacios (se puede leer la norma completa [aqui](#)).

Un algoritmo en Python **debe seguir esta estructura**:

```
sentencia
sentencia
...
estructura_control: # cada sentencia de control acaba en ":" 
    sentencia
    ...
    estructura_control:
        sentencia
        ...
        sentencia
        ...
    sentencia
```



No existe un límite en el número de sentencias que pueden aparecer en cada bloque (o nivel de sangrado). No obstante, siempre debe haber al menos una sentencia.

A continuación, se revisan las más importantes.

CONTINUAR

5.1. Condicionales

Las expresiones condicionales **facilitan la codificación de estructuras que bifurcan la ejecución del código en varias ramas o caminos de ejecución según condiciones definidas**. Existen varias formas.

5.1.1. IF SIMPLE

Tiene la estructura:

```
if expresión_booleana:  
    ejecutar código
```

Por ejemplo:

- Programa que muestra el mensaje “El número es positivo” si se escribe un **número mayor a cero**.

```
x = int(input("Escribe un número: "))  
if x > 0:  
    print ("El número es positivo")
```

Conclusión:

Si la expresión a continuación de la palabra **if** se evalúa como **True**, el bloque sangrado a continuación se ejecutará por completo. En caso de ser **False**, la ejecución continuará con la siguiente sentencia a continuación del **if en el mismo nivel de sangrado**.

CONTINUAR

5.1.2. IF-ELSE

La segunda forma de la sentencia **if** es la ejecución alternativa, en la cual existen **dos posibilidades y la condición determina cuál de ellas sería ejecutada**:

```
if expresión booleana:  
    ejecutar bloque 1  
else:  
    ejecutar bloque 2
```

Por ejemplo:

- Programa que solicita un número al usuario y muestra si está **dentro del rango 10-20**.

```
n = int(input('Escribe un número entero: ')) # entero
if(n>=10 and n<=20):
    print("Esta entre 10 y 20")
else:
    print("No está en ese rango")
```

- Programa que muestra el mensaje "El número es positivo" si se escribe **un número mayor que cero** y, en caso contrario, se muestra "El número es negativo".

```
x = int(input("Escribe un número: "))
if x > 0:
    print ("El número es positivo")
else:
    print ("El número es negativo")
```

Conclusión:

Dado que la condición debe ser obligatoriamente verdadera o falsa, **solo una de las alternativas será ejecutada**. Las alternativas reciben el nombre de ramas, ya que se trata de ramificaciones en el flujo de la ejecución.

CONTINUAR

5.1.3. IF-ELIF-ELSE

La tercera forma de la sentencia *if* es el condicional encadenado que permite que haya **más de dos posibilidades o ramas**:

```
if expresión booleana:
    ejecutar bloque 1
elif otra expresión booleana:
    ejecutar bloque 2
else:
    ejecutar por defecto
```

Por ejemplo:

- Programa que determina **si dos números son iguales**.

```
n1 = 10
n2 = 11
if n1 < n2:
    print (n2, " es mayor" )
elif n2 < n1:
    print (n1, " es mayor")
else:
    print ("Son iguales")
```

- Programa que determina **el mayor de tres números**.

```
n1 = 10
n2 = 11
n3 = 12
if n1 > n2 and n1 > n3:
    print (n1)
elif n2 > n1 and n2 > n3:
    print (n2)
elif n3 > n1 and n3 > n2:
    print (n3)
else:
    print ("Son iguales")
```

- Programa que determina **si una letra es vocal**.

```
x = input("Escribe una letra: ")
if x == "a" or x == "e" or x == "i" or x == "o" or x == "u":
    print(x, " es vocal minúscula")
elif x == "A" or x == "E" or x == "I" or x == "O" or x == "U":
    print(x, " es vocal mayúscula")
else:
    print(x, " no es vocal")
```

- Programa que muestra el mensaje "**El número es positivo**" si se escribe un **número mayor que cero**, "**El número es cero**" si el número es cero y "**El número es negativo**" si el **número es menor que cero**.

```
x = int(input("Escribe un número: "))
if x > 0:
    print ("El número es positivo")
elif x == 0:
    print ("El número es cero")
else:
    print ("El número es negativo")
```

Se puede observar que:

- No hay un límite para el número de sentencias elif. Si hay una cláusula else, debe ir al final, pero tampoco es obligatorio que esta exista.
- Cada condición es comprobada en orden. Si la primera es falsa, se comprueba la siguiente y así sucesivamente. Si una de ellas es verdadera, se ejecuta la rama correspondiente y la sentencia termina. Incluso si hay más de una condición que sea verdadera, solo se ejecuta la primera que se encuentra.

Saber más

Un condicional puede también estar anidado dentro de otro. Sin embargo, estos pueden ser difíciles de leer, por lo que deben evitarse y hay que tratar de usar operadores lógicos que permitan simplificar las sentencias condicionales anidadas.

[CONTINUAR](#)

5.2. Bucles

Los bucles **permiten la ejecución repetitiva de un bloque de sentencias**. Este bloque se ejecutará de manera secuencial mientras se cumplan las condiciones definidas en la estructura de control del bucle.

Tras cada iteración, se evaluará de nuevo la condición para determinar si se debe o no repetir la ejecución del bloque.

Para definir un bucle, se procede como sigue:

Se define la estructura de control del bucle, con la expresión que define las condiciones de ejecución.

Se construye el bloque de sentencias del bucle, con un nivel más de sangrado.

Cuando se define la última sentencia del bucle, se pueden añadir nuevas sentencias posteriores, fuera de este (al mismo nivel de sangrado que el bucle), para que sean ejecutadas cuando este finalice.

Existen varias formas:

CONTINUAR

5.2.1. WHILE

Los bucles de tipo while **permiten definir una estructura de control condicional**, generalmente una expresión booleana, de manera que las sentencias del bucle se ejecutarán indefinidamente mientras sea cierta la expresión (True).

Tras cada iteración, **la ejecución vuelve al inicio del bucle, donde se evalúa de nuevo la condición**. Solo cuando esta condición sea False, el programa saldrá del bucle y continuará por la siguiente sentencia en el mismo nivel de sangrado que el **while**.

```
while (expresión booleana):  
    código
```

Por ejemplo:

- Usando un while, programa que muestre los **números del 10 al 50**:

```
i = 10  
while i <=50 :  
    print(i)  
    i += 1
```

- Usando un while, programa que imprime **una palabra al revés**:

```
invertida = ""
cadena = "al revés"
cont = len(cadena)
indice = -1
while cont >= 1:
    invertida += cadena[indice]
    indice = indice + (-1)
    cont -= 1
print (invertida)
```

- Usando un while, programa que **calcula el factorial de un número**:

```
numero = 5
factorial = 1
print ("Factorial de",numero)
while numero > 0:
    factorial *= numero
    numero -= 1
print ("es ",factorial)
```

CONTINUAR

5.2.2. FOR

El siguiente tipo de bucle es el **for**. Se repite a través de un conjunto conocido de elementos, de modo que **ejecuta tantas iteraciones como elementos hay en el conjunto**. Es útil utilizar la función *range* para crear una secuencia de elementos sobre los que poder iterar. *Range* **puede tomar uno o dos valores**:

1

Si toma dos valores, genera todos los enteros desde la primera entrada hasta la segunda entrada-1. Por ejemplo: *range* (2, 5) = (2, 3, 4).

2

Y si toma un solo parámetro, entonces *range*(x) es equivalente a escribir *range*(0,x).

Tiene la siguiente estructura:

```
for variable in secuencia:
    código
```

Por ejemplo:

- Usando un for, programa que **muestra los números del 1 al 100**:

```
x = 101
for i in range(1, x):
    print(i)
```

- Usando un for, programa que **muestra los números pares del 1 al 100**:

```
for i in range(1, 101):
    if( (i%2) == 0 ):
        print(i)
```

- Usando un for, programa que pregunta al usuario cuántos números necesita validar. Posteriormente, el usuario teclea cada uno y **valida si son positivos, negativos o cero**:

```
total = int(input("¿Cuantos números vas a validar?"))
for y in range(0,total):
    x = int(input("Teclea un número: "))
    if x > 0:
        print ("El número es positivo")
    elif x == 0:
        print ("El número es cero")
    else:
        print ("El número es negativo")
```

- Un ejercicio más, programa que **imprime al revés la frase que escriba el usuario**:

```
entrada = (str(input("Teclea una frase: ")))
salida = ''
for n in entrada:
    salida = n + salida
print (salida)
```

Saber más

Téngase en cuenta que los bucles pueden estar anidados.

CONTINUAR

5.2.3. Terminación abrupta de un bucle: break

En ocasiones **puede ser necesario forzar la salida del bucle**, incluso aunque se sigan cumpliendo las condiciones definidas. Para ello, **en Python existe la sentencia `break`**, que termina inmediatamente la ejecución del bucle e indica al programa que debe continuar ejecutando la siguiente sentencia en el mismo nivel de sangrado que el bucle.

Por ejemplo, se puede construir un bucle `while` que solicite al usuario que introduzca palabras hasta que la palabra introducida sea "Fin":

```
while True:  
    linea = input ("Introduce Fin para finalizar: ")  
    if (linea == "Fin"):  
        break  
    print (linea)
```

```
while True:  
    linea = input ("Introduce Fin para finalizar: ")  
    if (linea == "Fin"):  
        break  
    print (linea)
```

Introduce Fin para finalizar: Test
Test
Introduce Fin para finalizar: Fin

Figura 23. Ejecución y salida del código.
Fuente: elaboración propia.

CONTINUAR

5.2.4. Interrumpir la iteración actual: continue

De forma similar a `break`, **se puede interrumpir en un punto concreto la ejecución de una iteración de un bucle, sin interrumpir el bucle completo**. Para ello Python ofrece la sentencia `continue`, que, tras ser ejecutada, **indica al programa que debe volver al inicio del bucle y reevaluar la condición para decidir si continúa o no la ejecución**. Las sentencias del bloque inmediatamente posteriores a `continue` no serán ejecutadas.

Por ejemplo, se puede extender el ejemplo anterior solicitando al usuario que introduzca solo palabras de tres caracteres como máximo, saltando aquellas de mayor longitud:

```
while True:  
    linea = input ("Introduce Fin para finalizar: ")  
    if len(linea) > 3:  
        continue  
    if (linea == "Fin"):  
        break  
    print (linea)
```



VI. Estructuras de datos

6.1. Tuplas

Una tupla es una secuencia de valores de cualquier tipo indexada por enteros. Las tuplas son inmutables —tienen una longitud fija y sus elementos no pueden cambiarse— y son comparables. Sintácticamente, una tupla es una lista de valores separados por comas y encerradas entre paréntesis.

Ejemplo

```
t = ("a","b","c","d","e","f")
```

Para crear una tupla **con un único elemento**, es necesario **incluir una coma al final**.

```
p = (4,)
```

Otra forma de construir una tupla es usar la función interna *tuple*, que **crea una tupla vacía si se invoca sin argumentos** y, si se le proporciona como argumento una secuencia (cadena, lista o tupla), genera una tupla con los elementos de la secuencia. Por ejemplo:

```
t = tuple()  
print (t)  
t = tuple("supercalifragilisticoespialidoso")  
print (t)
```

```
t=tuple ()  
print (t)
```

```
()
```

```
t=tuple("supercalifrastilisticoespidalidoso")  
print (t)
```

```
('s', 'u', 'p', 'e', 'r', 'c', 'a', 'l', 'i', 'f', 'r', 'a', 's', 't', 'i', 'l', 'i', 's', 't',  
'i', 'c', 'o', 'e', 's', 'p', 'i', 'd', 'a', 'l', 'i', 'd', 'o', 's', 'o')
```

Figura 24. Ejecución y salida del código.

Fuente: elaboración propia.

CONTINUAR

Los principales operadores sobre tuplas son:

- El operador corchete permite acceder a un elemento mediante su posición o índice.

```
t = (3, 5, "c", "d", "e")
print ( t[0] ) #3

t=(1, "Enero", 2021)
t[0] # Devuelve 1
t[1] # Devuelve Enero
t[2] # Devuelve 2019
t[-1] # Devuelve el último elemento: 2021
```

- El operador **slice** selecciona un rango de elementos.

```
t = (3, 5, "c", "d", "e")
print ( t[1:3] ) # (5, 'c')
```

- No se pueden modificar los elementos de una tupla, pero se puede reemplazar una tupla por otra.

```
t[1]=45
```

```
-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-61-05771ac3f9cb> in <module>()  
----> 1 t[1]=45
```

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

Figura 25. Error al tratar de modificar los elementos de una tupla.
Fuente: elaboración propia

- La función *len()* calcula la longitud de una tupla.

```
t = (2021, "Alondra", "Hacker", (1, "Enero", 2021))
len(t)                                # Devuelve 4
len(t[3])                             # Devuelve 3
```

- Se pueden comparar dos tuplas usando los caracteres de comparación ya vistos. De forma nativa, los comparadores comparan elemento a elemento de cada tupla. Se comienza comparando el primer elemento de cada secuencia. Si es igual en ambas, pasa al siguiente elemento y así sucesivamente hasta que encuentra uno que es diferente. A partir de ese momento, los elementos siguientes ya no se tienen en cuenta.

```
(0,1,2) < (0,1,2)          # False
(0,1,2) == (0,1,2)         # True
(0,1,3) == (0,1,2)         # False
(0,1,3) < (0,1,2)          # False
(0,1,3) > (0,1,2)          # True
```

CONTINUAR

—

Algunos ejemplos con tuplas:

- **Tupla con los nombres de los meses del año.** Se solicita al usuario que escriba un número y se muestra el mes del año que corresponde:

```
meses = ("Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio", "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre")
numero = int(input("Escribe un número entre 1 y 12: "))
if(numero >= 1 and numero <= 12):
    print("El mes ", numero, "es: ", meses[numero-1])
else:
    print(":(")
```

- **Tupla con números aleatorios del 1 al 10.** Se solicita un número al usuario y se imprime cuantas veces se repite:

```
tupla = (2,3,2,1,3,4,7,8,9,7,6,5,5,6,7,8,9,10,4,3,2,1,6,4,5,3,6,6,6,6,6)
numero = int(input("Escribe un numero del 1 al 10: "))

contador= 0
for i in tupla:
    if numero == i:
        contador = contador + 1

print ("El número ", numero, " se repite ", contador, " veces.")
```

CONTINUAR

6.2. Listas

Al igual que las tuplas, **las listas son secuencias de elementos de cualquier tipo, indexados mediante números enteros.** La principal diferencia con las tuplas es que las listas sí son mutables. Es decir, su contenido puede cambiarse, así como su longitud (se pueden añadir y borrar elementos).

El método más simple para crear una lista es **encerrar los elementos entre corchetes**. Por ejemplo:

```
t = [10, 20, 30, 40]
```

La asignación de valores a una lista no retorna nada; sin embargo, **si se usa el nombre de la lista, es posible ver el contenido de la variable.** Por ejemplo:

```
t = [10, 20, 30, 40]
t                                # Devuelve [10, 20, 30, 40]
```

Una lista que no contiene elementos recibe el nombre de lista vacía —**se crea con unos corchetes vacíos []**—. Por ejemplo:

```
t = []
t                                # Devuelve []
```

Para acceder a los elementos de una lista, se usa el operador corchete, que contiene una expresión que especifica el índice —los índices comienzan por 0—. **Los índices de una lista se caracterizan por:**

- Cualquier expresión entera** puede utilizarse como índice.
- Si se intenta leer o escribir un elemento que no existe, **se obtiene un IndexError**.
- Si un índice tiene un valor negativo, **se cuenta hacia atrás desde el final de la lista.**

Por ejemplo:

```
t = [10, 20, 30, 40]
t[2]                      # Devuelve 30
t[-2]                     # Devuelve 30
```

CONTINUAR

Recuerda:

Como se ha comentado, **las listas son mutables**, puesto que su estructura puede cambiarse después de haberlas creado. Por ejemplo:

```
numeros = [17, 123]
numeros[1] = 5
print (numeros)           # Devuelve [17, 5]
```

Los elementos en una lista **no tienen por qué ser todos del mismo tipo**. Por ejemplo:

```
t = ["casa", 3.0, 5, [11, 20]]
```

Cuando una lista está dentro de otra, se dice que está **anidada**. En una lista anidada, cada lista interna solo cuenta como un único elemento. Por ejemplo:

```
t = ["casa", 3.0, 5, [11, 20]]
```

CONTINUAR

6.2.1. Operadores y funciones

Las listas tienen definidos los siguientes operadores y funciones.

Operadores:

- El operador `+` concatena listas.

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print (c)                      # Devuelve [1, 2, 3, 4, 5, 6]
```

- El operador `in` permite preguntar la pertenencia de un elemento a una lista.

```
a = [1, 2, 3]
2 in a                  # True
5 in a                  # False
```

El operador `in` se puede usar **para recorrer los elementos de una lista usando un bucle for**, por ejemplo:

```
a = [1, 2, 3]
for x in a:
    print (x)
```

- El operador `*` repite una lista el número especificado de veces.

```
[1, 2, 3] * 3    # Devuelve [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

- El operador (**slice**) cuya sintaxis es `[inicio:final:salto]` permite seleccionar secciones de una lista.

```
t = [1, 2, 3, [4, 5, 6]]
t[1:3]                  # Devuelve [2, 3]
t[2:4]                  # Devuelve [3, [4, 5, 6]]
t[:]
t[1:]                  # Devuelve [1, 2, 3, [4, 5, 6]]
# Devuelve [2, 3, [4, 5, 6]]
```

- El operador `del` elimina un elemento de la lista referenciado en forma de índice.

```
t = [1, 2, 3, [4, 5, 6]]  
del t[1]  
t  
# Devuelve [1, 3, [4, 5, 6]]
```

CONTINUAR

Funciones

- La función `sum()` permite realizar la suma de una lista de números.

```
t = [1, 2, 3, 4, 5, 6]  
sum (t) # Devuelve 21
```

`t=[1,2,3,4]
sum(t)`

10

Figura 26. Ejecución y salida del código.

Fuente: elaboración propia.

- Las funciones `max()` y `min()` proporcionan el elemento máximo/mínimo de una lista.

```
t = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
print ( max(t), min (t) ) # Devuelve 9 1
```

- La función `len()` proporciona la longitud de una lista.

```
t = [1, 2, 3, 4, 5, 6, 7, 8, 9]
len(t)                      # Devuelve 9
```

- La función `range()` crea una secuencia de valores a partir del dado como parámetro. Es útil para los bucles de tipo `for`. Por ejemplo:

```
lista = range (-3,3)
for i in lista:
    print (i)                  # Devuelve -3 -2 -1 0 1 2
```

- Para ver el contenido generado por `range()`, se debe usar el constructor `list`.

```
lista = range (-3,3)
list ( lista )                # Devuelve [-3, -2, -1, 0, 1, 2]
```

- `append` añade un nuevo elemento al final de una lista.

```
t = [1, 2, 3, [4, 5, 6]]
t.append("nuevo")
t                         # Devuelve [1, 2, 3, [4, 5, 6], 'nuevo']
```

- `extend` toma una lista como argumento y añade al final de la actual todos sus elementos.

```
t1 = [1, 2, 3, 4, 5, 6]
t2 = [7, 8, 9]
t1.extend(t2)
t1                         # Devuelve [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- `sort` ordena los elementos de una lista de menor a mayor.

```
t = ["d", "e", "a", "c", "x"]
t.sort()
t                         # Devuelve ['a', 'c', 'd', 'e', 'x']
t = [7, 6, 5, 4, 3, 2, 1]
```

```
t.sort()  
t # Devuelve [1, 2, 3, 4, 5, 6, 7]
```

- El método *pop* elimina un elemento de la lista referenciado en forma de índice. Devuelve el elemento que ha sido eliminado. Si no se proporciona un índice, borra y devuelve el último elemento.

```
t = ["d", "e", "a", "c", "x"]  
x = t.pop(1)  
print (t) # Devuelve ['d', 'a', 'c', 'x']  
print (x) # Devuelve e
```

- El método *remove* permite eliminar un elemento de la lista referenciándolo por su valor.

```
t = ["d", "e", "a", "c", "x"]  
t.remove("e")  
print (t) # Devuelve ['d', 'a', 'c', 'x']
```

Saber más

Se puede ver más acerca de manipulación de listas en la [documentación oficial de Python](#).

CONTINUAR

6.2.2. Equivalencia en las listas

En Python, **dos listas son equivalentes si tienen los mismos elementos, pero no son idénticas**. Sin embargo, **si dos listas son idénticas** (es decir, sus referencias apuntan al mismo “objeto” en memoria, de manera que, si se cambia una, el cambio se refleja en la otra), **también son equivalentes**.

Por tanto, la **equivalencia no implica que sean idénticas**. Para comprobar si dos variables son idénticas, se puede usar el operador **is**.

- En este ejemplo, **a** y **b** son equivalentes, pero no idénticas:

```
a = [1, 2, 3]
b = [1, 2, 3]
a is b          # Devuelve false
```

- En este ejemplo, **a** y **b** son idénticas.

```
a = [1, 2, 3]
a = b
a is b          # Devuelve true
```

- Si **a** y **b** son idénticas, significa que la lista tiene dos referencias o nombres diferentes. Así, los cambios que se hagan usando cualquiera de los nombres afectan a la misma lista.

```
a = [1, 2, 3]
b = a
b[0] = 45
print (a)      # Devuelve [45, 2, 3]
```

Hay que tener en cuenta que:

- En las operaciones que se realizan sobre las listas existen aquellas que modifican listas y otras que crean listas nuevas. Por ejemplo, el método *append* modifica una lista, pero el operador + crea una lista nueva.

```
t1 = [2, 3]
t2 = [5]
t1.append(4)
print ( t1, t1+t2 )      # Devuelve [2, 3, 4] [2, 3, 4, 5]
```

- La mayoría de los métodos modifican la lista y devuelven el valor **None**.

CONTINUAR

Ejemplos con listas

- Programa que solicita al usuario una palabra y guarda los caracteres en una lista sin repetirlos:

```
palabra = input("Escribe una palabra: ")
lista = []
for c in palabra:
    if(c not in lista):
        lista.append(c)
print(palabra)
print(lista)
```

- Lista vacía, se solicita al usuario valores para la lista. Al final, se muestra la suma y el promedio:

```
lista = []
total = int(input('¿Cuántos elementos tendrá la lista?: '))
for i in range(total):
    numero = int(input("Escribe un numero: "))
    lista.append(numero)
suma = sum(lista)
promedio = suma / len(lista)
print (lista)
print("La suma es ",suma)
print("El promedio es ",promedio)
```

CONTINUAR

6.2.3. Listas por comprensión

Una lista por comprensión **es una expresión compacta para definir listas, conjuntos y diccionarios en Python usando estructuras de control iterativas.** Se trata de definir cada uno de los elementos sin tener que nombrar cada uno de ellos. La forma general es:

```
[expresión for val in <colección> if <condición>]
```

Ejemplos:

```
lista = [x*2 for x in [3, 4, 5]]  
lista # Devuelve [6, 8, 10]
```

```
lista = [x+5 for x in [3,4,5] if x > 3]  
lista # Devuelve [9, 10]
```

```
lista = [ x for x,y in [(1,2),(3,4),(5,6)]]  
lista # Devuelve [1, 3, 5]
```

Ejemplos con listas por comprensión:

- Lista que contiene los valores de 3 elevado a la x , donde x es un número par y pertenece al rango del 1 a 100 :

```
c = [3 ** x for x in range(1,101) if x % 2 == 0]  
print (c)
```

- Lista con los múltiplos de 3 , entre un rango de 1 a 100 :

```
c = [ x for x in range(1,101) if x % 3 == 0]  
print (c)
```

CONTINUAR

6.3. Diccionarios

Un diccionario es una colección no ordenada de pares clave-valor donde la clave y el valor son objetos que pueden ser de (casi) cualquier tipo. Los diccionarios permiten acceder y manipular valores concretos a través de sus claves.

La función `dict()` crea un diccionario nuevo sin elementos.

```
ejemplo = dict()  
ejemplo # Devuelve {}
```

 Las llaves `{}` representan un diccionario vacío.

CONTINUAR

Para añadir elementos al diccionario, **se pueden usar corchetes y usar acceso indexado a través de la clave**. Por ejemplo:

```
ejemplo = dict()  
ejemplo # Devuelve {}  
ejemplo["primero"] = "Libro"  
ejemplo # Devuelve {'primero': 'Libro'}
```

Otra forma de crear un diccionario es **mediante una secuencia de pares clave-valor separados por comas y encerrados entre llaves**. Por ejemplo:

```
ejemplo2 = {"primero": "Libro", "segundo": 34, "tercero": (3, 4)}  
ejemplo2 # Devuelve {'primero': 'Libro',  
'segundo': 34, 'tercero': (3, 4)}
```

El orden de los elementos en un diccionario es impredecible, pero eso no es importante, ya que se usan las claves para buscar los valores correspondientes. En este sentido, si la clave especificada no está en el diccionario, se obtiene una excepción de tipo **KeyError**.

Algunos métodos:

Función len —

La función `len` devuelve el número de parejas clave-valor.

```
ejemplo2 = {"primero": "Libro", "segundo": 34, "tercero": (3,4)}
len (ejemplo2) # 3
```

Operador in —

El operador **in** permite comprobar si un valor existe como clave dentro del diccionario.

```
ejemplo2 = {"primero": "Libro", "segundo": 34, "tercero": (3,4)}
"primero" in ejemplo2 # True
```

Método values —

Para ver si algo aparece como valor en un diccionario, se puede usar el método **values**, que devuelve los valores como una lista, y después usar el operador **in** sobre esa lista.

```
valores = ejemplo2.values()
"uno" in valores # False
```

Método get () —

El método **get()** toma una clave y un valor por defecto. Si la clave aparece en el diccionario **get**, devuelve el valor correspondiente. En caso contrario, devuelve el valor por defecto.

```
contadores = {"naranjas":1, "limones":42, "peras":100}
print(contadores.get("uvas",0)) # Devuelve 0
print(contadores.get("naranjas",0)) # Devuelve 1
```

Método keys () —

El método **keys()** crea una lista con las claves de un diccionario.

```
contadores = {"naranjas":1, "limones":42, "peras":100}
contadores.keys() # dict_keys(['naranjas', 'limones', 'peras'])
```

Método items () —

El método **items ()** devuelve una lista de tuplas, cada una de las cuales es una pareja clave-valor sin ningún orden definido.

```
contadores = {"naranjas":1, "limones":42, "peras":100}
t = contadores.items()
t           # dict_keys([('naranjas', 1), ('limones', 42), ('peras', 100)])
```

CONTINUAR

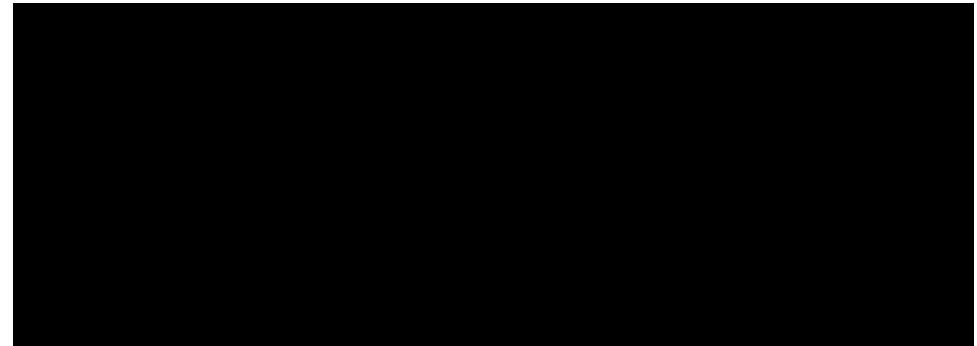
Se debe tener en cuenta lo siguiente:

Se puede utilizar un diccionario como una secuencia en una sentencia *for*, de manera que se recorren todas las claves del diccionario. Por ejemplo:

```
diccionario = {1:"hola",2:42, 3:100}
for clave in diccionario:
    print (clave, diccionario[clave])    # 1 hola
# 2 42
# 3 100
```

El ejemplo anterior se podría haber realizado de una manera equivalente utilizando el método **ítems ()**, que devuelve una lista de tuplas donde cada tupla contiene dos elementos: clave y valor.

```
diccionario = {1:"hola", 2:42, 3:100}
for clave, valor in diccionario.items():
    print ( clave, valor )                 # 1 hola
# 2 42
# 3 100
```



VII. Funciones

Una función es una **secuencia de sentencias** que realizan una operación y **que reciben un nombre**.

Sus principales características son:

- Cuando se define una función, **se especifica el nombre y la secuencia de sentencias**.
- Una vez que se ha definido una función, **se puede llamar a la función por ese nombre** y reutilizarla a lo largo del programa.
- El resultado de la función se llama **valor de retorno**. Las funciones pueden retornar uno o varios valores, aunque no es obligatorio que lo hagan.

Proceso:

Para crear una función se utiliza la palabra reservada **def**. A continuación, aparece el nombre de la función, entre paréntesis los parámetros, y finaliza con **:**. Esta línea se denomina cabecera de la función. El código de la función, o cuerpo, se define en líneas sucesivas, con un nivel más de sangrado con respecto a la cabecera. El cuerpo puede contener cualquier número de sentencias. Para devolver el valor se usa la palabra reservada **return**.

Por ejemplo:



Función que suma tres números y retorna el resultado:

```
#función que suma tres números y devuelve el resultado
def suma_tres(x, y, z): # 3 argumentos posicionales
    m1 = x + y
    m2 = m1 + z
    return m2
```



Función que multiplica los números de una lista:

```
def multiplicacion (lista):
    resultado = 1
    for i in lista:
        resultado *= i
    return (resultado)
lista = {1,2,3,4}
print (multiplicacion ( lista ))      # Devuelve 24
```



Función que determina si una palabra es palíndroma¹¹:

```
def palindromo(palabra):
    palabra_al_reves = palabra[::-1]
    print(palabra_al_reves)
    if( palabra == palabra_al_reves ):
        print("Es palindromo")
    else:
        print("No es palindromo")
palindromo("Aguila")           # 'NO es palindromo'
palindromo("arenera")          # 'es palindromo'
```

¹¹[Palindrome Definition.](#)

CONTINUAR

Las reglas para los nombres de las funciones son las mismas que para las variables: **se pueden usar letras, números y algunos signos de puntuación, pero el primer carácter no puede ser un número**. No se puede usar una palabra clave como nombre de una función y se debería evitar también tener una variable y una función con el mismo nombre. Las funciones con paréntesis vacíos después del nombre indican que esta función no toma ningún argumento.

La sintaxis para llamar a una función definida consiste en indicar el nombre de la función junto a una expresión entre paréntesis denominados argumentos de la función. El argumento es un valor o variable que se pasa a la función como parámetro de entrada.

```
r = suma_tres( 1, 2, 3 )      # Devuelve 6 y se lo asigna a r
r = suma_tres( 4, 5, 6 )      # Devuelve 15 y se lo asigna a r
```

Algunas características:

- La definición de una función debe ser ejecutada antes de que la función se llame por primera vez, y no generan ninguna salida. Sin embargo, las sentencias dentro de cada función son ejecutadas solamente cuando se llama a esa función.
- En las funciones no se especifica el tipo de parámetro ni lo que se retorna. No obstante, en Python 3 se introdujo una nueva sintaxis que permite indicar el tipo “esperado” de cada parámetro, en la cabecera de la definición de la función. Y se subraya “esperado” porque el intérprete no evalúa el tipo de cada parámetro en tiempo de ejecución. Se trata simplemente de una convención para facilitar la comprensión del código y, por tanto, está constituida como una buena práctica en el ámbito del desarrollo en Python.¹²

¹²Say yes more. Yes opens doors. No closes them. Yes pushes us. No keeps us safe at home. Imagine all the opportunities waiting for a yes.

- Las definiciones de funciones no alteran el flujo de la ejecución de un programa debido a que las sentencias dentro de una función no son ejecutadas. Sin embargo, una llamada a una función es como un desvío en el flujo de la ejecución. En vez de pasar a la siguiente sentencia, el flujo salta al cuerpo de la función, ejecuta todas las sentencias que hay allí y después vuelve al punto donde lo dejó.
- Los argumentos de una función solo existen en el ámbito de esta, y dejan de hacerlo cuando la ejecución de la función termina. Asimismo, se puede pasar como parámetro cualquier expresión, que será evaluada justo antes de iniciar la ejecución de la función, y su resultado quedará almacenado en el argumento correspondiente.

En el inicio de la ejecución, los parámetros proporcionados en la invocación pasan a ser referenciados por los argumentos definidos en la cabecera de esta, para poder ser usados en el cuerpo:

```
def sumandos(a, b):
    suma = a+b
    return suma
c = 5
x = sumandos(c, c+7) # dentro de la función, c pasa a ser a y c+7 pasa a ser b. Una vez finalizada la ejecución, a
print (x)           # Devuelve 17
```

- Cuando se definen los argumentos de una función, estos pueden tener valores por defecto, que se utilizarán en caso de que se invoque la función sin esos argumentos. Por ejemplo:

```
def ejemplo(a=3):
    print (a)
ejemplo()           # Devuelve 3
```

- Una vez que se ha definido una función, puede usarse dentro de otra, lo cual facilita la descomposición de un problema y su resolución mediante una combinación de llamadas a funciones. Por ejemplo:

```
def ejemplo1 (a, b):
    return a+b
def ejemplo2 (a,b,c):
    return c+ ejemplo1(a,b)
ejemplo2(3,4,5)      # Devuelve 12
```

CONTINUAR

Hay dos tipos de funciones:

Aquellas que producen resultados, con los que se querrá hacer algo.

Como asignárselo a una variable.

```
def ejemplo1(a):
    return 3 * a
b = ejemplo1(4)
print(b)                      # Devuelve 12
```

Aquellas que realizan alguna acción, pero no devuelven un valor.

Si se asigna el resultado a una variable, se obtiene el valor None.

```
def ejemplo1 (a):
    print( 3*a)
b = ejemplo1(4)          #Devuelve      12
print (b)                #Devuelve      None
```

CONTINUAR

Es posible **definir funciones que devuelvan múltiples resultados**, los cuales se empaquetan en una tupla. Por ejemplo, función que recibe el número de segundos y calcula la duración de horas, minutos y segundos:

```

def segundos_a_HMS(segundos):
    hs = int(segundos / 3600)
    min = int((segundos % 3600) / 60)
    seg = int((segundos % 3600 ) % 60)
    return (hs, min, seg)

(h, m, s) = segundos_a_HMS(3661)
print ("Son",h,"horas",m,"minutos",s,"segundos")
# Son 1 horas 1 minutos 1 segundos

```

Python proporciona un número importante de funciones internas que pueden usarse sin necesidad de tener que definirlas previamente. Se denominan ***built-in functions***¹³:

- Las funciones *max* y *min* dan respectivamente el valor mayor y menor de una lista.
- La función *len* devuelve cuantos elementos hay en su argumento. Si el argumento es una cadena, devuelve el número de caracteres que hay en la cadena.
- Funciones que permiten convertir valores de un tipo a otro: *int()*, *float()*, y *str()*.
- La función *chr* tiene como parámetro un número entero que devuelve un carácter (cadena) cuyo código Unicode¹⁴ es el número entero (parámetro). Por ejemplo, *chr(97)* devuelve el carácter (cadena) 'a'.
- La función *ord* tiene como parámetro un carácter (cadena) que devuelve el valor ASCII de una cadena. Por ejemplo, *ord ('a')* devuelve el entero 97.

¹³[Built-in functions](#).

¹⁴[Unicode in Python](#).

CONTINUAR

En Python, **el paso de argumentos a una función se hace por referencia**, de manera que las modificaciones que se hagan sobre los argumentos se mantienen después de la llamada y ejecución de la función, dentro de las variables proporcionadas como parámetro en la llamada.

Ejemplos de programas que utilizan funciones internas:



Imprime el abecedario en minúsculas basado en el código Unicode:

```
for x in range(97,122):
    print ("Carácter UNICODE de ",x , '=' ,chr(x) )
```



Imprime el abecedario en mayúsculas basado en el código Unicode:

```
for x in range(65,91):
    print ("Carácter UNICODE de ",x , '=' ,chr(x) )
```



VIII. Excepciones

Durante la ejecución de un programa, se pueden producir situaciones inesperadas que den lugar a un error. Son situaciones en las que, por norma general, la continuación de la ejecución del programa de forma normal produciría resultados incorrectos.

Por eso, en esas ocasiones, es conveniente “lanzar” una excepción, de manera que se interrumpa la ejecución de las sentencias sucesivas de ese bloque y los bloques exteriores hasta llegar a un bloque “padre” que disponga de lógica apropiada para gestionar ese error.

Por tanto, en este momento es conveniente destacar que:

- Una excepción se puede lanzar de forma voluntaria en cualquier punto de un programa, si se dan condiciones ante las que no se debe continuar la ejecución.
- Las excepciones se pueden capturar, para tratarse y, en caso de ser posible, proporcionar un flujo alternativo de ejecución.
- Existen diferentes tipos de excepciones dentro del sistema y todas parten de una misma clase de error: el error de tipo **Exception**. Se pueden definir otros tipos de excepciones, pero eso queda fuera del alcance de esta unidad.

CONTINUAR

8.1. Lanzamiento de excepciones

Las excepciones se lanzan usando la sentencia *raise*. Hay que hacerlo cuando se detecte una inconsistencia o situación de anomalía en el programa. Por ejemplo:

```
animal = input("Elije tu animal preferido entre caballo, perro, gato:")
if animal not in ['caballo', 'perro', 'gato']:
    throw Exception("Animal incorrecto ")
print("Tu elección ha sido", animal) # Solo se imprime si
animal está entre los 3 definidos.
```

8.2. Captura de excepciones

Pueden capturarse dentro de expresiones de tipo **try-except**. Estas expresiones definen dos tipos de bloques:

- **try** contiene el código que potencialmente podría lanzar una excepción.

- **except** define el bloque que se debe ejecutar en caso de que se produzca una excepción dentro del bloque **try**.
- Se pueden definir varios bloques **except** para un mismo **try**, de manera que se puedan tratar de forma diferente los diferentes tipos de excepción que se pueden lanzar en el **try**.

Sintaxis

```
try:  
    animal = input("Elije tu animal preferido entre caballo, perro, gato:")  
    if animal not in ['caballo', 'perro', 'gato']:  
        throw Exception("Animal incorrecto")  
        print("Tu elección ha sido", animal)  
    except Exception as e: # Guardamos en la variable e la  
    excepción lanzada  
        print("Elección incorrecta, ", e) # Imprimiría "Elección  
incorrecta, Animal incorrecto"
```

IX. Importación de módulos

Python dispone de una amplia variedad de módulos o librerías. Los módulos son programas que amplían las funciones y clases de Python para realizar tareas específicas. Los módulos tienen extensión py. En la [página web de Python](#), se puede encontrar el índice de módulos de Python.

Para poder utilizarlas, **hay que importarlas previamente**, lo cual se puede hacer de varias formas:

- Importar todo el módulo mediante la palabra reservada **import**, de manera que para utilizar un elemento hay que usar el nombre del módulo seguido de un punto (.) y el nombre del elemento que se desee obtener.

```
import random
for i in range(4):
    x = random.random()
    print (x)
```

- Importar solo algunos elementos del módulo mediante la estructura **from nombre_modulo import lista_elementos**, de manera que los elementos importados se usan directamente por su nombre. Por ejemplo, programa que genera cuatro números enteros aleatorios entre 1 y 10:

```
from random import randint
for i in range(4):
    x = random.randint(1,10)
    print (x)
```

Programa que calcula el perímetro o área de un círculo. El usuario debe teclear el radio del círculo y seleccionar el cálculo que necesita:

```
from math import pi
radio = float(input('Teclea el radio de un círculo: '))
print ("Selecciona una opción")
print ("1 Calcular perímetro")
print ("2 Calcular área")
print ("3 Ambos")
print ("4 Salir")
opcion = int(input('Teclea opción: '))
while not opcion in (1,2,3,4):
    opcion = int(input('Teclea opción: '))
if opcion == 1:
    print ("Perímetro es: ",(2*pi*radio) )
elif opcion == 2:
    print ("Área es: ",(pi*radio*radio) )
elif opcion == 3:
    print ("Perímetro es: ",(2*pi*radio) )
```

```
    print ("Área es: ",(pi*radio*radio) )
else:
    print ("Adios")
```

- Importar todo el módulo mediante la palabra reservada **import** y definir un alias mediante la palabra reservada **as**, de manera que para usar un elemento hay que utilizar el nombre del módulo seguido de un punto () y el nombre del elemento que se deseé obtener. Por ejemplo, usando el módulo **os**¹⁵, programa que imprime el directorio actual y su contenido:

```
import os as te
print("Directorio actual: ", te.getcwd())
print("Directorios/ficheros: ", te.listdir(os.getcwd()))
```

¹⁵Página web ["Files and Directories"](#).



Para conocer las operaciones disponibles de un módulo, se puede usar el comando **dir**.

```
import os
dir (os)
```

X. Gestión de archivos

Para abrir un archivo en Python se usa la función `open(fichero, modo)`, donde:

FICHERO

MODO

Es una cadena que contiene la ruta completa a un fichero, incluido su nombre.

FICHERO

MODO

Es otra cadena que indicará el modo de apertura del fichero. Los modos más destacados son:

- 'r' solo lectura.
- 'w' solo escritura. Si el fichero tuviera contenido, lo elimina.
- 'a' adición. Se escribe al final del fichero, respetando su contenido anterior.

Saber más

La lista completa de modos se puede consultar en la [documentación oficial](#).

La función `open` retorna un objeto `file`, que expone una serie de métodos para manipular el descriptor de fichero asociado.

La operación más sencilla que se puede realizar sobre un archivo es leer su contenido. Para procesarlo línea por línea, es posible hacerlo de la siguiente forma:

```
fichero = open("cuna.txt")
for linea in fichero:
```

```
print(linea)  
fichero.close()
```

CONTINUAR

Al terminar de trabajar con un archivo, se debe cerrar. Esto es especialmente importante cuando el fichero se abre en modo edición (**w** o **a**), ya que, según el sistema, podrían no guardarse todos los cambios una vez finalizada la ejecución del programa. Para ello, se usa *close*.

Ya te vemos dormida.

Tu barca es de madera por la orilla.

Blanca princesa de nunca.

Duerme por la noche oscura.

Cuerpo y tierra de nieve.

Duerme por el alba, duerme.

Ya te alejas dormida.

Figura 27. Lectura del contenido de un archivo.

Fuente: elaboración propia.

```
fichero.close()
```

Además, usando la función `readlines` es posible recuperar de una sola vez todo el contenido del archivo estructurado en forma de líneas.

```
['Ya te vemos dormida. \n',
 'Tu barca es de madera por la orilla. \n',
 'Blanca princesa de nunca. \n',
 'Duerme por la noche oscura.\n',
 'Cuerpo y tierra de nieve. \n',
 'Duerme por el alba, duerme.\n',
 'Ya te alejas dormida. ']
```

Figura 28. Función `readlines`.

Fuente: elaboración propia.

```
fichero= open("cuna.txt")
lineas = fichero.readlines()
print(lineas)
```

En este caso, la variable `lineas` tendrá una lista de cadenas con todas las líneas del archivo. Téngase en cuenta que es posible eliminar los saltos de línea.

```
lineas[0].rstrip()
```

CONTINUAR

Para abrir un fichero en modo escritura:



Crear un archivo llamado “prueba.txt” y guardar el texto ‘Primer archivo’:

```
f = open ('prueba.txt', 'w')
f.write('Primer archivo')
f.close()
```



Crear un archivo llamado “nuevo.txt” y guardar las líneas pares del archivo “cuna.txt” que se puede descargar en este [enlace](#):



cuna.txt.zip

649 B



```
arc_write = open ('nuevo.txt', 'w')
for i, line in enumerate(lineas):
    if i%2 == 0:
        arc_write.write( str(i) + ' ' + line )
    else:
        pass
arc_write.close()
```

Nuevo fichero llamado “nuevo.txt”:

0 Ya te vemos dormida.
2 Blanca princesa de nunca.
4 Cuerpo y tierra de nieve. Duerme por el alba, duerme.

Figura 29. Líneas pares del archivo “cuna.txt”.

Fuente: elaboración propia.

En ocasiones, **es posible que un bloque de código termine de forma abrupta** (mediante un error o excepción). Esto puede ser especialmente grave si ocurre en mitad de una operación de escritura en un fichero abierto. En este caso, en el evento de una excepción durante la escritura de un dato en un fichero, haría que la excepción se elevara a la rutina llamante, evitando que se ejecute la función `close`. Como se ha visto, **esto podría causar que todo o parte del contenido añadido al fichero se perdiera**.

Para evitarlo, **Python ofrece un mecanismo de protección denominado context managers¹⁷**, que permite utilizar los recursos justo en el momento en el que son necesarios.

¹⁷[Context Managers](#).

CONTINUAR

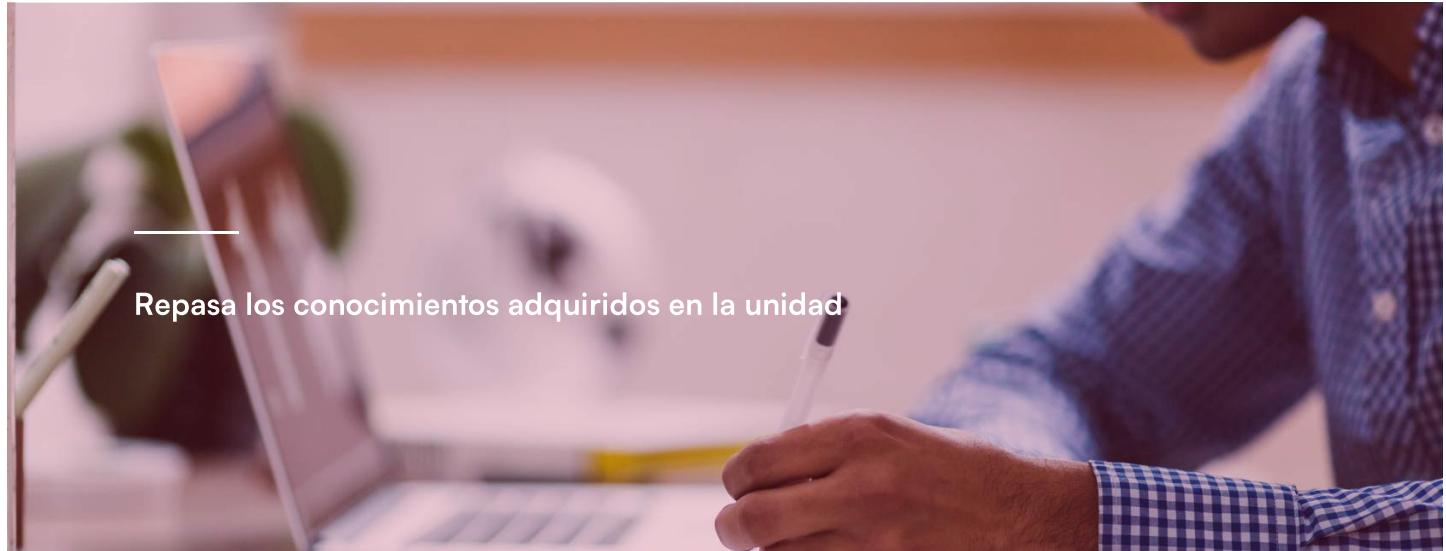
Uno de los más utilizados está disponible a través de la sentencia *with*, que permite “encapsular” el acceso a un fichero, asegurando que se cierra en caso de errores inesperados:

```
with open ('prueba.txt','w') as f:  
    f.write('Primer archivo')
```

Esto es equivalente a escribir:

```
f = open ('prueba.txt','w')  
try:  
    f.write('Primer archivo')  
finally:  
    f.close()
```

XI. Resumen



Repasa los conocimientos adquiridos en la unidad

En esta unidad, se ha introducido el lenguaje de programación *Python*. Se trata de un lenguaje de propósito general que dispone de potentes librerías para análisis de datos, lo cual lo convierte en uno de los lenguajes más utilizados en el ámbito del *big data*.

Asimismo, se ha mostrado cómo instalar y utilizar la herramienta Jupyter Notebook. Esta herramienta permite desarrollar y ejecutar código Python de manera muy visual, alternando código con secciones de lenguaje de marcas (Markup language). De esta manera, se pueden componer análisis completos donde intercalar gráficas, imágenes, vídeos, así como cualquier otro componente web junto con el código.

Esto convierte a Jupyter Notebook en una de las herramientas preferidas por analistas y científicos de datos para presentar informes interactivos de cierta complejidad.

Gracias a Jupyter, se han podido explorar cómodamente, y de forma interactiva, las funcionalidades básicas de Python.

Se ha hecho un recorrido por los primeros pasos dentro del lenguaje, aprendiendo a asignar variables y a realizar operaciones aritméticas básicas.

Se ha aprendido que Python es un lenguaje de tipado dinámico, lo que implica que una variable puede contener datos de cualquier tipo y que esos tipos se definen en tiempo de ejecución.

Se ha enseñado cómo utilizar las estructuras básicas de control de flujo en Python, como *if*-*else*, *for* y *while*, y se ha visto la peculiar sintaxis de Python a la hora de definir bloques de código, mediante sangrado.

Se ha trabajado con las estructuras de datos básicas del lenguaje, como las listas y los diccionarios. Es importante recordar que una lista es una colección de elementos accesibles a través de un índice, mientras que un diccionario es una colección de datos referenciados por una clave.

Asimismo, se ha aprendido a trabajar con funciones y a entender su vital importancia a la hora de definir secciones de código reutilizable.

Se ha hecho hincapié en la potencia de la sintaxis de Python a la hora de facilitar la construcción de estructuras de datos complejas mediante listas por comprensión. En una sola línea se pueden construir listas que en otros lenguajes implicarían el desarrollo de varios bloques anidados.

Por último, se ha visto el sistema de gestión de errores de Python, basado en excepciones. Las excepciones son un mecanismo muy habitual en los lenguajes de programación actuales y permiten detener la ejecución del algoritmo en un punto en el que se produce una situación inesperada o errónea.

La ejecución de los ejercicios de la unidad se muestra en el cuaderno que se puede descargar en el siguiente enlace: UD2_Python.ipynb. Se recomienda abrir el cuaderno con Jupyter Notebook, basta con iniciar el servicio "Jupyter Notebook" e importar el fichero "UD2_Python.ipynb" desde la opción *Upload*.



UD2_Python.zip
69.3 KB



XII. Caso práctico con solución



Aplica los conocimientos adquiridos en esta unidad

ENUNCIADO

Considérese el siguiente número de mil dígitos:

73167176531330624919225119674426574742355349194934
96983520312774506326239578318016984801869478851843
85861560789112949495459501737958331952853208805511
12540698747158523863050715693290963295227443043557
66896648950445244523161731856403098711121722383113
62229893423380308135336276614282806444486645238749
30358907296290491560440772390713810515859307960866
70172427121883998797908792274921901699720888093776
65727333001053367881220235421809751254540594752243
52584907711670556013604839586446706324415722155397
53697817977846174064955149290862569321978468622482
83972241375657056057490261407972968652414535100474
82166370484403199890008895243450658541227588666881
16427171479924442928230863465674813919123162824586
17866458359124566529476545682848912883142607690042
2421902267105562632111109370544217506941658960408
07198403850962455444362981230987879927244284909188
84580156166097919133875499200524063689912560717606

05886116467109405077541002256983155200055935729725
71636269561882670428252483600823257530420752963450

Se sabe que la secuencia de cuatro dígitos consecutivos de este número que computa el mayor producto es $9 \times 9 \times 8 \times 9 = 5832$.

DATOS

Desarrollar una función que, dado un número de dígitos consecutivos, obtenga la secuencia que calcule el mayor producto.

¿Cuál sería la secuencia de 30 dígitos que cumple este requisito?

[VER SOLUCIÓN](#)

SOLUCIÓN

```
def largest_product(number, n):
    max_prod = 0
    numbers = ''
    for i in range(len(number) - n):
        prod = 1
        for x in number[i:i+n]:
            prod *= int(x)
        if prod > max_prod:
            max_prod = prod
            numbers = number[i:i+n]
    return numbers, max_prod
```

XIII. Glosario



El glosario contiene términos destacados para la comprensión de la unidad

Programa —

Se trata de un conjunto de sentencias de un lenguaje de programación cuya ejecución produce un resultado.

Entorno de desarrollo —

También denominado editor. Es una aplicación informática que permite crear y ejecutar programas para un determinado lenguaje de programación.

Jupyter Notebook —

Entorno de desarrollo para Python.

Notebook —

Tipo de archivo utilizado por Jupyter Notebook para editar código Python que permite integrar otros recursos como vídeos o imágenes.

Variable —

Nombre que referencia una posición de memoria.

Expresión —

Conjunto de valores o variables combinadas mediante un conjunto de operadores que representan un valor de un tipo de datos.

Estructura de control —

Hace referencia a sentencias de un lenguaje que permiten estructurar el flujo de ejecución de un programa.

Bucle —

Estructura de control de Python que permite repetir un conjunto de acciones un número de veces que depende de una condición determinada.

Condicional —

Estructura de control de Python que permite ejecutar acciones alternativas de acuerdo con el valor de una expresión.

Estructura de datos —

Hace referencia a una forma lógica determinada de almacenar la información.

Mutabilidad —

Es una propiedad de las estructuras de datos en Python que indica si los datos pueden modificar su estructura y contenido una vez que se han definido.

Diccionario —

Estructura de datos de Python que permite almacenar la información en forma de parejas clave-valor. Es un tipo de datos mutable.

Lista —

Estructura de datos de Python que permite almacenar un conjunto de datos de diferentes tipos. Es un tipo de datos mutable.

Tupla —

Estructura de datos de Python que permite almacenar un conjunto de datos de diferentes tipos. Es similar a las listas, pero no mutable.

Función —

Es la unidad de estructuración de un programa en Python. Representa un conjunto de acciones que reciben un nombre y que pueden depender de un conjunto de parámetros y devolver como resultado uno o más valores.

Listas por comprensión —

Es una forma de definir una lista en la que, en vez de indicar los valores concretos, se indica la expresión que permite generar dichos valores.

Módulo —

También denominado librería, se trata de un conjunto de funciones o métodos que añaden nuevas funcionalidades a Python respecto a las que tiene de manera estándar.

Excepción —

Situación de error que interrumpe la ejecución del programa en un punto dado, elevando el problema en la jerarquía del algoritmo hasta que un bloque la capture para gestionarla.

XIV. Bibliografía

- Bahit, E. *Python para principiantes*. The Oxford Research Centre, 2020.
- Marzal Varó, A., Gracia Luengo, I. y García Sevilla, P. [Introducción a la programación con Python 3](#). Castelló de la Plana: Publicacions de la Universitat Jaume I. Servei de Comunicació i Publicacions; 2014.
- Rossum, G. [Guía de aprendizaje de Python. Release 2.4.1a0](#). Fred L. Drake, Jr. 2005
- Rossum, G. y Drake, F. L. Jr. [Referencia de la Biblioteca de Python](#). 2000.
- Wachenchauzer, R., Manterola, M., Curia, M., Medrano, M. y Páez, N. [Algoritmos de programación con Python](#). 2006-2019.