



we design and
create
professional
quality software

Language - Agnostic

Language-agnostic programming

La programación o secuencias de comandos independientes del lenguaje es un paradigma de desarrollo de software en el que se elige un lenguaje en particular debido a su idoneidad para una tarea en particular, y no simplemente por el conjunto de habilidades disponibles dentro de un equipo de desarrollo.

Modern Portfolio Designed

Te quedo? Listo.
(Rodrigo Bueno)





Características de PYTHON.

INSTALACION Y ENTORNO
GESTION DE PAQUETES
GESTION DE ENTORNOS VIRTUALES
TIPOS DE DATOS
CONTROL DE FLUJO
COLECCIONES
BUCLES
FUNCIONES
RECURSIVIDAD
FUNCIONES LAMBDA
PROGRAMACION FUNCIONAL
EXCEPCIONES
ARCHIVOS
MODULOS Y PAQUETES
TESTING & FRAMEWORK TEST





PYTHON

INSTALACION Y ENTORNO

INSTALACION Y ENTORNO EN PYTHON.

Descarga e instalación

Es primordial para comenzar a desarrollar con Python es tenerlo instalado en nuestra computadora. Veamos cómo conseguirlo.

¿Tengo Python instalado?

En algunos sistemas operativos, como las distribuciones de Linux y Mac OS, Python estará instalado de fábrica por cuanto es utilizado por herramientas del sistema.

Se puede comprobar abriendo una [terminal](#) y escribiendo `python --version` (en Linux/Mac) o `py --version` (en Windows). Usaremos la versión de Python 3, . Si tienes una versión de Python 2 es necesario que instales una más nueva (varias instalaciones pueden coexistir en un mismo sistema). Si obtienes un error, entonces Python no está instalado; será mejor pasar al siguiente apartado.

No tengo Python instalado

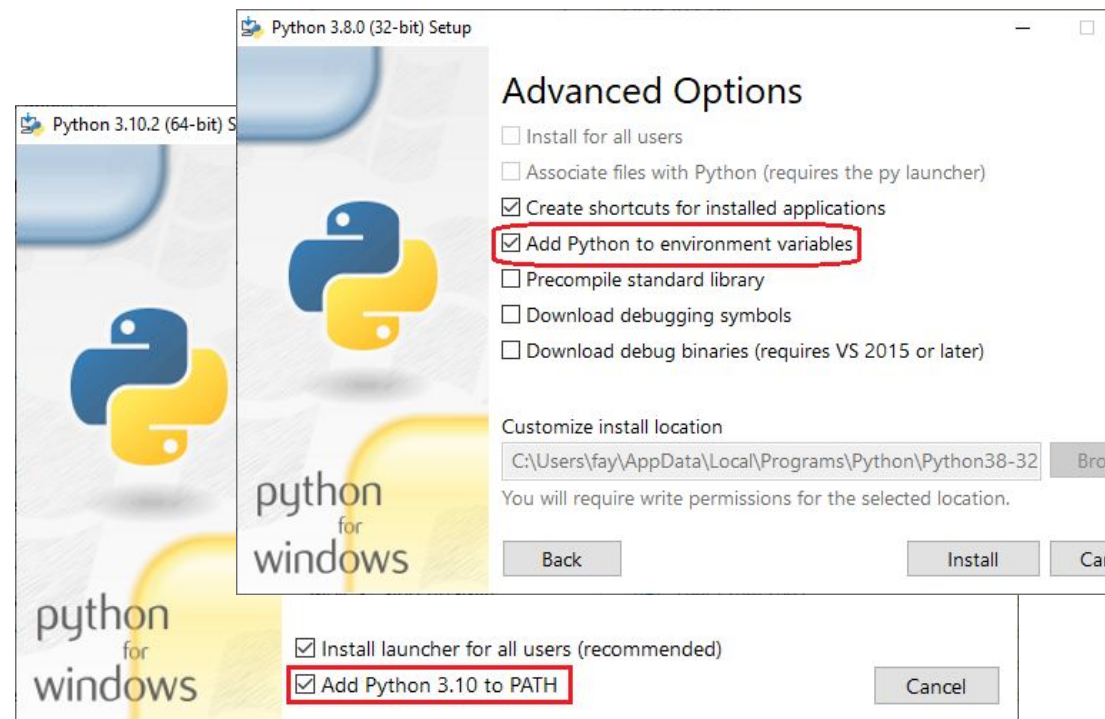
Para aquellos que no tienen Python instalado de fábrica, vayamos a <https://www.python.org/> para descargar la última versión disponible.



The screenshot shows the Python.org website with the 'Downloads' tab selected. Under 'All releases', the 'Python 3.10.2' link is highlighted with a red box. The page also includes a 'Download for Windows' section with a note that Python 3.9+ cannot be used on Windows 7 or earlier.

En Windows, obtendremos un feliz instalador que hará todo el trabajo por nosotros. En lo único que debemos prestar atención es en seleccionar el casillero Add Python X.Y.Z to PATH, que nos permitirá abrir Python sin indicar su ruta completa.

Opcionalmente puedes elegir en dónde quieres que se instale Python (presionando en Customize installation). Por practicidad, generalmente se instala —en Windows— en C:\PythonXY\ (considerando que C:\ es la unidad principal).



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



Primer Script en Python - ¡Hola, mundo!

La instalación de Python incluye un IDLE (un programa en donde escribir el código) llamado IDLE, muy básico, pero que nos será útil en nuestros primeros pasos en el lenguaje. Si ya tienes tu editor de código favorito también puedes usarlo.

Para crear nuestro primer programa vamos a abrir IDLE y seleccionar el menú File > New File para crear un nuevo documento. Luego, escribiremos lo siguiente.

```
print("Hola Mundo")
```

Para poder ejecutar este pequeño código primero debemos guardarlo. Para ello, en IDLE vamos a ir al menú File > Save y lo guardaremos en el escritorio como **hola.py**.



Este es un auténtico código de Python. ¡Solo una línea! En ella llamamos a la función incorporada `print()` y le pasamos una cadena de caracteres como argumento para que imprima en la pantalla. Se dice que es incorporada ya que es una herramienta que el lenguaje nos pone siempre a disposición en nuestros programas. Existen muchas otras que iremos conociendo en el camino

Ahora bien, recordemos que Python es un lenguaje **interpretado**. Esto quiere decir que no hay un programa compilador que transforme nuestro código fuente (`hola.py`) y lo convierta en un archivo ejecutable (`hola.exe`, por ejemplo); más bien, hay un programa llamado **intérprete** al cual le indicamos que queremos ejecutar un archivo determinado. Todos los editores de código pueden hacer esto automáticamente (por ejemplo, en IDLE, presionando F5), no obstante, en este tutorial vamos a hacerlo de la forma manual, esto es, invocando al intérprete desde la terminal. Esto nos dará un panorama más amplio sobre cómo funciona todo en el mundo de Python.

Acceso al [IDLE](#)

Acceso al [IDLE](#)
en MS Windows

Start PowerShell
[wt.bat](#) WSL Linux

```
#include <stdio.h>
int main(void)
{
    printf("Hello, world!");
}
```

C

```
#include <iostream.h>
int main()
{
    std::cout << "Hello, world! ";
    return 0;
}
```

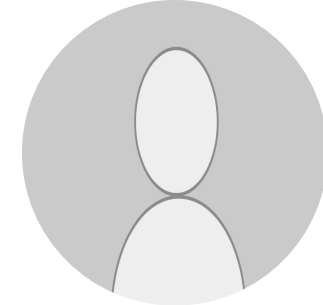
C++

```
class HelloWorld {
    public static void main(String[]
args) {
        System.out.println("Hello,
World!");
    }
}
```

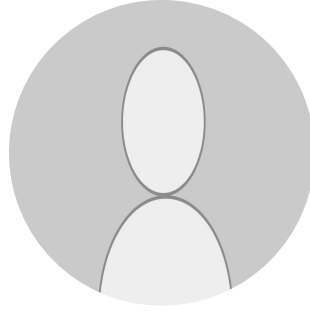
Java

```
print "Hello, world!"
```

Python



¡Hola, mundo!



Entonces, como decíamos, vamos a abrir la terminal.

Todo sistema operativo tiene algún atajo para esto. En Windows, puedes presionar CTRL + R y escribir cmd, o bien buscar el programa de nombre "Símbolo del sistema". El primer paso es ubicarnos en la ruta en donde hemos guardado nuestro archivo (el escritorio) vía el comando cd. Hecho esto, ejecutamos nuestro script de Python escribiendo python o py seguido del nombre del archivo.

En Linux/Mac:

```
> cd Desktop  
> python3 hola.py  
¡Hola, mundo!
```

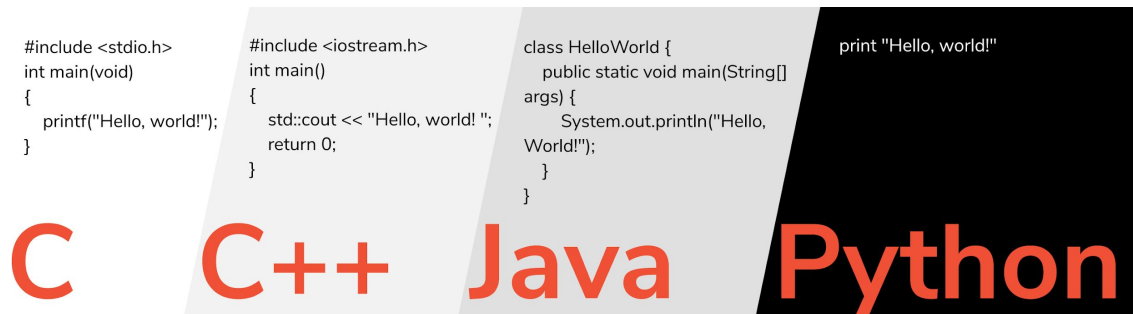
En Windows:

```
> cd Desktop  
> py hola.py  
¡Hola, mundo!
```



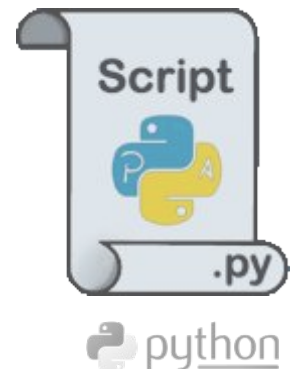
Hemos ejecutado tu primer código de Python.

Recuerda que los programas que escribimos en Python son por defecto aplicaciones de consola. Haciendo doble clic sobre hola.py hará que el intérprete ejecute nuestro archivo, pero una vez impreso el mensaje se cerrará automáticamente (pues es lo que ocurre con todo programa cuando alcanza la última línea de código).



Start PowerShell
wt.bat WSL Linux

Acceso al IIDLE
en MS Windows





Consola interactiva >>>

La consola interactiva es una herramienta que nos permite escribir sentencias de código de Python al mismo tiempo que se ejecutan. La estaremos usando todo el tiempo, ya que es ideal para probar pequeñas porciones de código. Una vez que la cerramos, todo lo que hemos escrito en ella se pierde.

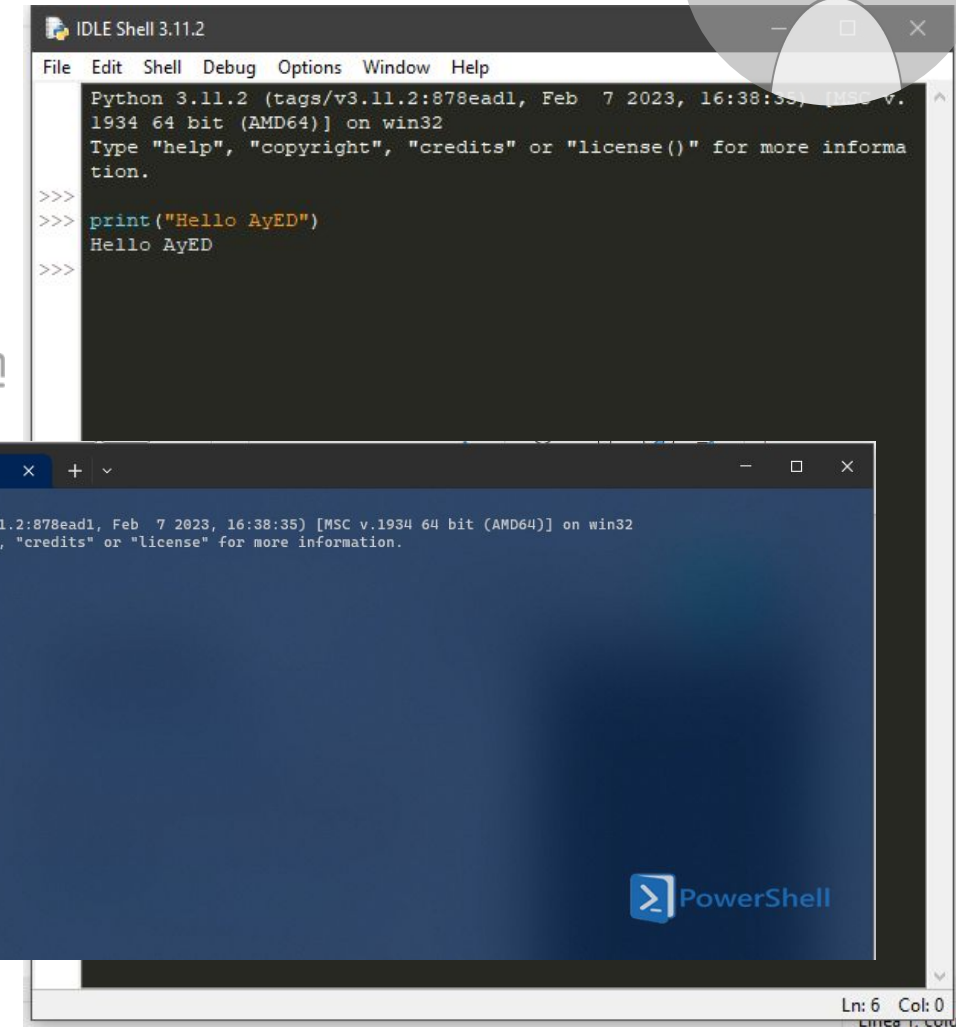
Para iniciarla vamos a escribir el comando **python** (en Linux/Mac) o **py** (en Windows). Es el mismo que hemos estado utilizando anteriormente, a excepción de que cuando no le indicamos ningún archivo para ejecutar, entiende que queremos iniciar la consola interactiva. Al abrirla deberías ver algo similar a lo siguiente.

```
C:\Users\Administrador>py
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Hagamos la prueba de escribir una sentencia que ya conocemos y luego presionemos enter.

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

Como se observa, el código es ejecutado una vez presionada la tecla enter: en este caso, imprime un mensaje en la pantalla. A partir de ahora, todo código que comience con >>> indica que debe escribirse en la consola interactiva.



Acceso al [IDLE](#)

Acceso a la [Consola](#)
`python -m idlelib`

Start PowerShell
[wt.bat](#) WSL Linux

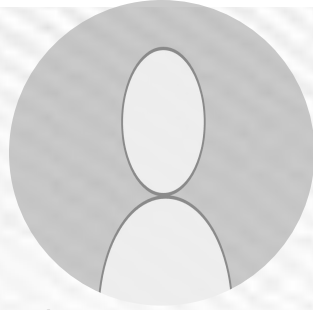


PYTHON

GESTION DE PAQUETES

INTRODUCCION A LA GESTION D EPAQUETES PYTHON.

Gestión de dependencias en Python con pip



Un tema que todo desarrollador Python debe aprender tarde o temprano es saber gestionar correctamente las dependencias de sus programas. Para ello Python nos proporciona la herramienta **pip**

¿En qué consiste la gestión de dependencias?

En Python, tanto programas grandes como scripts se suelen desarrollar apoyándose en librerías y/o frameworks de terceros a los cuales llamamos dependencias. A su vez estas dependencias pueden tener otras dependencias llamadas dependencias transitivas. Esto sucede, por ejemplo, con pandas. Pandas es una librería muy popular pensada para manipular y analizar datos, cuyo desarrollo está basado en NumPy, otra librería muy conocida para realizar cálculos matriciales. Por tanto, al utilizar pandas también utilizamos, aunque de forma indirecta, NumPy.

Gestionar todas las dependencias de un programa grande de forma manual puede llegar a ser un proceso tedioso. Por un lado requiere dedicarle tiempo y por otro se trata de un proceso en el cual es fácil realizar fallos. La solución a estos inconvenientes son los llamados sistemas de gestión de paquetes.

Antes de continuar, puntualizar que en Python cuando decimos “paquetes” solemos referirnos a librerías y frameworks de terceros. Algunos ejemplos muy conocidos son, por ejemplo, Django, flask para el desarrollo web, o Requests para realizar peticiones HTTP

Python tiene su propio sistema de gestión de paquetes llamado pip. Esta aplicación es en realidad una interfaz de línea de comandos que viene preinstalada con cualquier versión moderna de Python.

Para saber si lo tenemos instalado es tan sencillo como ejecutar el siguiente comando en nuestro terminal.

\$ pip --version

En caso que así sea, nos informará sobre la versión de pip que tenemos instalada. En la siguiente imagen podemos ver que en mi computadora tengo instalada la versión 20.3.4 de pip, la cual es ejecutada por Python 3.9.

```
Administrador: Símbolo del sistema
Microsoft Windows [Versión 10.0.19044.2364]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Administrador>pip --version
pip 22.3.1 from E:\code\python\Lib\site-packages\pip (python 3.11)

C:\Users\Administrador>
C:\Users\Administrador>
```

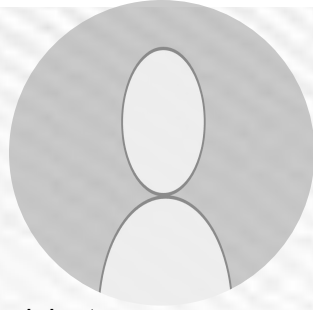
Acceso al [IDLE](#)

Language Tips:

Adicionalmente, también podemos ver un listado de todos los argumentos admitidos por pip mediante el siguiente comando.
\$ pip --help



Instalar y actualizar pip



- **Instalación**

Como he comentado en el apartado anterior, pip está incluido por defecto en versiones modernas de Python. En concreto se introdujo a partir de Python 2.7.9 en adelante (en Python 2) y a partir de Python 3.4 en adelante (en Python 3). Si tu versión de Python es anterior a las mencionadas existen dos alternativas para poder instalarlo. La primera, aunque sea un poco obvia hay que mencionarla, y es que actualices a una versión de Python más moderna. Si por el motivo que sea esto no fuera posible, alternativamente puedes añadir pip a tu instalación de Python, pero ten en cuenta que a partir de pip 21.0 se eliminó el soporte para Python 2.

En Windows o macOS, para añadir pip a la instalación de Python, primero tienes que descargar el script de instalación de pip. Luego, ejecutarlo como cualquier otro script.

```
$ python get-pip.py
```

En distribuciones de Linux (Debian/Ubuntu) el proceso varía un poco ya que hay que usar su sistema de gestión de paquetes.

```
$ sudo apt update
```

```
$ sudo apt install python3-pip
```

- **Actualización**

El proceso de actualización de pip varía en función de tu sistema operativo. En Windows y macOS se utiliza directamente el comando pip, mientras que en Linux la actualización se realiza con el sistema de gestión de paquetes. A continuación te detallo como actualizar pip para cada uno de estos tres sistemas operativos.

Windows

```
C:\>pip install --upgrade pip setuptools
```

macOS

```
$ python -m pip install --upgrade pip
```

Linux

```
$ sudo apt update && sudo apt upgrade python3-pip
```

Acceso al [IDLE](#)

Start PowerShell
[wt.bat](#) WSL Linux



Repositorios de paquetes de Python



- **Repositorios de paquetes de Python**

El repositorio oficial (y más grande) de paquetes de Python es el [Python Package Index \(PyPI\)](https://www.pypi.org/). Cualquier programador puede registrarse gratuitamente en PyPI y subir ahí sus propios paquetes. Una vez que un paquete aparece en PyPI, cualquier persona puede instalarlo localmente mediante pip. Pero como no hay ningún proceso de revisión ni de aseguramiento de la calidad, es recomendado invertir algo de tiempo en revisar el software que estamos adquiriendo.

En PyPI podemos realizar búsquedas de paquetes por nombre y/o filtrar resultados en función de ciertos criterios como el estado de desarrollo, la licencia, etc. Además, cada paquete tiene su propia página en la web de PyPI. Ahí podemos obtener mucha información sobre un paquete en cuestión: la versión, la licencia, el comando para instalarlo, el nombre del autor, las versiones de Python con las que se ha testeado el paquete, y mucho más. Para que te hagas una idea de ello puedes consultar la página de Requests en PyPI.

<https://www.pypi.org/>

- **Instalar dependencias con pip**

Para instalar con pip cualquier paquete disponible en PyPI, simplemente debes teclear el siguiente comando:

```
$ pip install nombre
```

Donde nombre es el nombre del paquete deseado. La herramienta pip no sólo instalará el paquete que le indicamos y su dependencias, sino que además los almacenará en una caché local de caras a futuras instalaciones.

Otro comando útil, complementario al de instalar, es el que nos muestra un listado de los paquetes instalados que es el siguiente:

```
$ pip list
```

Una consideración a tener en cuenta es que pip instala por defecto los paquetes en el entorno global de Python. Sin embargo, es recomendado instalar nuestras dependencias en entornos virtuales de Python, ya que de este modo mantenemos las dependencias separadas por proyectos y evitamos posibles conflictos entre versiones de una misma dependencia.

Acceso al [IDLE](#)



Repositorios de paquetes de Python

- **Instalación de versiones anteriores**

Por defecto pip instala la última versión disponible de un paquete. Sin embargo, también es posible instalar versiones específicas. Para ello tenemos que especificar la versión deseada como se muestra a continuación, donde instalamos la versión 2.23.0 de la librería Requests:

```
$ pip install requests==2.23.0
```

Otra posibilidad es instalar versiones que supongan actualizaciones menores de una versión específica para, por ejemplo, aprovechar la corrección de un bug. El siguiente comando instala la versión 2.18.4 de la librería Requests, que es la más reciente de las versiones 2.18.X.

```
$ pip install requests~=2.18.0
```

- **Instalación desde GitHub**

Aunque lo más recomendado es instalar las dependencias directamente desde PyPI, también lo podemos hacer desde sus repositorios en GitHub. Para ello basta con ejecutar un comando tal que así:

```
$ pip install git+https://github.com/usuario/repositorio.git@rama
```

De este modo, podemos instalar la versión de la rama máster de Requests con el siguiente comando:

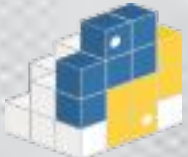
```
$ pip install git+https://github.com/kennethreitz/requests.git@master
```

Además, también podemos instalar commits o versiones específicas indicando respectivamente el hash o el número de versión.

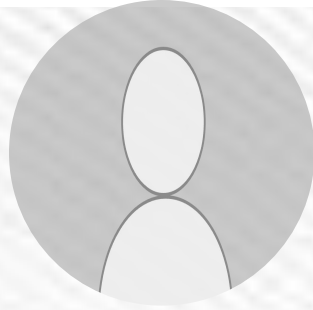


Start PowerShell
[wt.bat](#) WSL Linux

Acceso al [IDLE](#)



Repositorios de paquetes de Python



- **Obtener información de un paquete instalado**

Una vez hemos instalado un paquete en nuestro entorno de Python, podemos obtener información adicional sobre el mismo en nuestro terminal. Para ello tenemos que ejecutar el siguiente comando:

```
$ pip show nombre
```



- **Identificar y actualizar dependencias obsoletas**

Con el tiempo, las librerías que tenemos instaladas en nuestro entorno van recibiendo actualizaciones y se quedan obsoletas. Para ver un listado de las librerías que disponen de una nueva versión puedes ejecutar el siguiente comando:

```
$ pip list --outdated
```

, **Desinstalar dependencias**

En algunas ocasiones instalamos una librería simplemente para probarla. Si queremos desinstalarla de nuestro entorno, es tan sencillo como ejecutar el siguiente comando:

```
$ pip uninstall nombre
```

Hemos de tener en cuenta que este comando no desinstala las posibles dependencias transitivas. Por eso son tan útiles los entornos virtuales, ya que nos permiten eliminar todas las dependencias de golpe eliminando la carpeta del entorno virtual

- **Conclusiones**

En este apartado hemos visto a fondo todo lo que concierne a la **gestión de dependencias en Python con pip**. Desde cómo instalar y actualizar esta herramienta, a cómo utilizarla para instalar, listar, actualizar y eliminar las dependencias que vamos a utilizar para desarrollar nuestros programas o scripts en Python. Otro aspecto que hemos visto es la **importancia de combinar pip con entornos virtuales** para mantener nuestro entorno global de Python limpio, y poder evitar posibles conflictos entre versiones. Veamos ahora cómo gestionar entornos virtuales en Python.

Acceso al [IDLE](#)



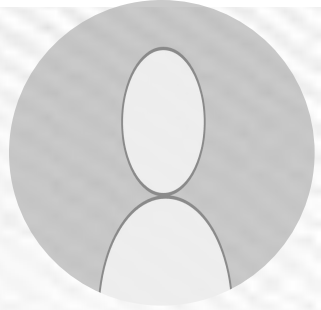
PYTHON

GESTION DE ENTORNOS VIRTUALES

ADMINISTRAR ENTORNOS VIRTUALES EN PYTHON.



Crear Entornos Virtuales en Python

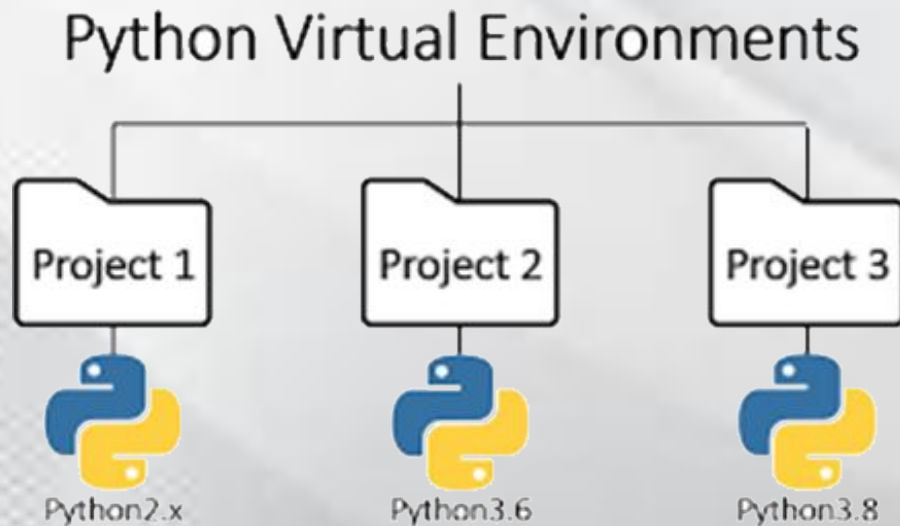


Un entorno virtual es un espacio donde podemos instalar paquetes específicos para un proyecto. Es decir, permite tener para un proyecto determinado un conjunto de paquetes/librerías aislados de la instalación principal de Python en nuestro sistema. Una razón para usar entornos virtuales es si por ejemplo tenemos muchos proyectos que utilizan una versión específica de una librería, pero queremos probar una versión más reciente de dicha librería sin crear errores en nuestros proyectos existentes.

Para la creación de entornos virtuales se recomienda utilizar el módulo **venv**, el cual viene instalado por defecto con la librería estándar de Python desde la versión 3.3. Por otro lado, la instalación de paquetes se realiza mediante la herramienta pip, la cual viene incluida por defecto a partir de Python 3.4. Como los comandos que vamos a ver en este post utilizan estas dos herramientas, se asume que se trabaja como mínimo con la versión 3.4.

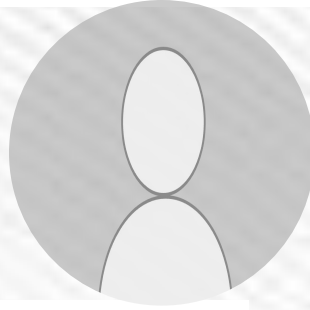
Índice

- Lista de comandos
- Comandos relacionados con la gestión de entornos virtuales
- Comandos relacionados con la gestión de paquetes
- Buenas prácticas con los entornos virtuales



Acceso al [IDLE](#)

Crear Entornos Virtuales en Python



Lista de comandos | Los siguientes comandos son válidos para el símbolo del sistema (CMD) de Windows, y la aplicación de terminal de macOS y Linux.

Comandos relacionados con la gestión de entornos virtuales

1. Crear un entorno virtual nuevo

En Windows, y asumiendo que Python está incluido dentro de las variables del sistema:

```
C:\>python -m venv c:\ruta\al\entorno\virtual
```

En macOS y Linux:

```
$ python3 -m venv ruta/al/entorno/virtual
```

Es recomendado que la carpeta para el entorno virtual sea una subcarpeta del proyecto Python al que esta asociado.

2. Activar un entorno virtual

En Windows:

```
C:\>c:\ruta\al\entorno\virtual\scripts\activate.bat
```

En macOS y Linux:

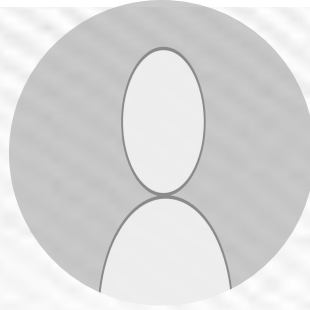
```
$ source ruta/al/entorno/virtual/bin/activate
```

Sea cual sea nuestro sistema operativo sabremos que el entorno virtual se ha activado porque su nombre aparece entre paréntesis delante del prompt.



Acceso al [IDLE](#)

Crear Entornos Virtuales en Python



Lista de comandos | Los siguientes comandos son válidos para el símbolo del sistema (CMD) de Windows, y la aplicación de terminal de macOS y Linux.

Comandos relacionados con la gestión de entornos virtuales

3. Desactivar un entorno virtual

Este comando es idéntico para Windows, macOS y Linux:

```
$ deactivate
```

4. Eliminar un entorno virtual

En Windows:

```
C:\>rmdir c:\ruta\al\entorno\virtual /s
```

En macOS y Linux:

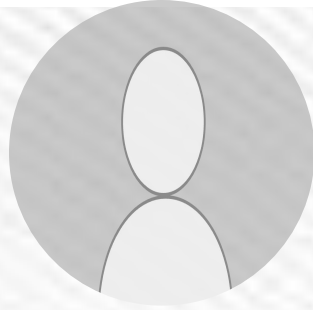
```
$ rm -rf ruta/al/entorno/virtual
```

Eliminar un entorno virtual es tan sencillo como eliminar la carpeta que lo contiene. Por ello, esta operación también se puede realizar desde el correspondiente administrador de archivos.



Acceso al [IDLE](#)

Comandos relacionados con la gestión de paquetes



La gestión de los paquetes instalados, sea en el entorno global de Python o en uno virtual, se realiza mediante la herramienta pip. Esta herramienta obtiene los paquetes que le mandamos instalar del Python Package Index. Para que los siguientes comandos funcionen bajo Windows, se asume que Python está incluido en las variables del sistema.

Instalar una nuevo paquete

`$ pip install paquete`

Siendo paquete el nombre del paquete/librería a instalar. Este comando instala también automáticamente las dependencias del paquete que deseamos instalar.

Eliminar un paquete

`$ pip uninstall paquete`

Siendo paquete el nombre del paquete/librería a desinstalar. Este comando no desinstala cualquier dependencia del paquete que desinstalamos.

Listar los paquetes instalados

`$ pip list`

Existe otro comando que lista los paquetes en formato del archivo requirements.txt. Este archivo contiene un listado de las versiones de los paquetes necesarios para ejecutar un proyecto en Python.

`$ pip freeze`

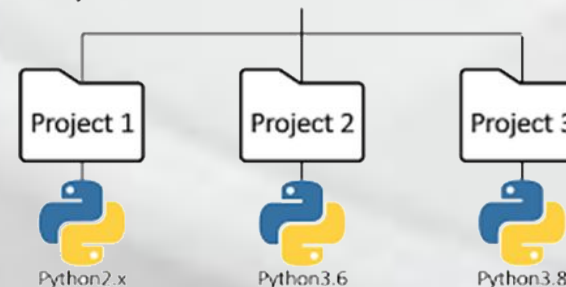
En sistemas Windows podemos copiar la salida de este comando, pegarla en un editor de notas como Atom o Sublime Text (que también podemos usar como nuestro entorno de programación), y guardarla manualmente con el nombre requirements.txt. Sin embargo, si usamos macOS o Linux podemos guardar la salida directamente en un fichero de texto.

`$ pip freeze > requirements.txt`

Instalar los paquetes del fichero requirements.txt

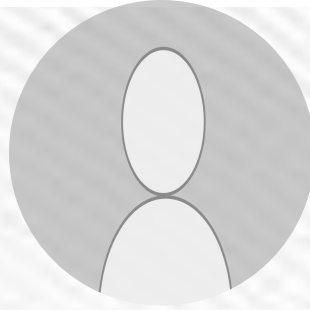
`$ pip install -r requirements.txt`

Python Virtual Environments



Acceso al [IDLE](#)

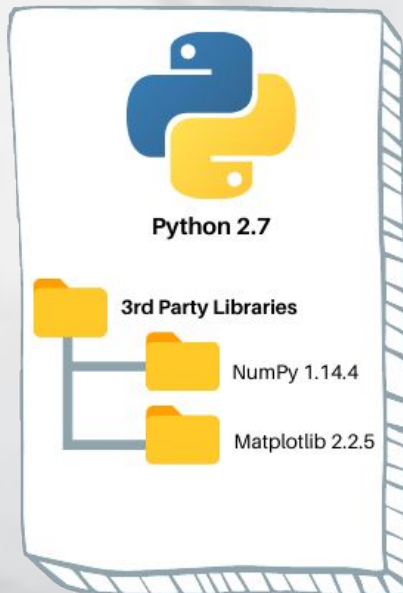
Buenas prácticas con los entornos virtuales



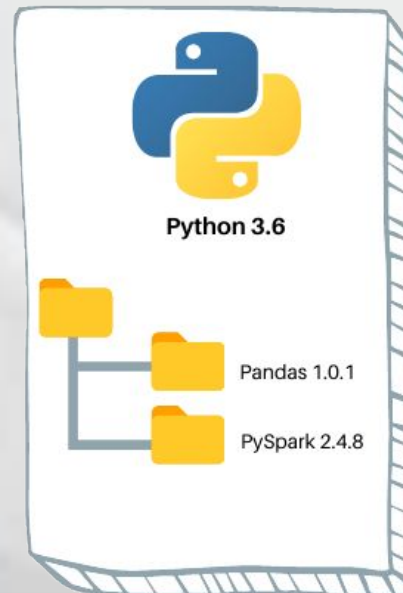
Existen dos consideraciones adicionales a tener en cuenta respecto a los entornos virtuales:

- No se debe añadir manualmente ningún fichero dentro de una carpeta que almacena un entorno virtual. Esta carpeta sólo debe contener los ficheros que se crean por defecto al crear el entorno virtual y los paquetes adicionales que hayamos instalado con el comando pip.
- La carpeta del entorno virtual no debe incluirse a nuestra herramienta de control de versiones. En lugar de ello es recomendado añadir un fichero requirements.txt con el listado de los paquetes necesarios para ejecutar el proyecto.

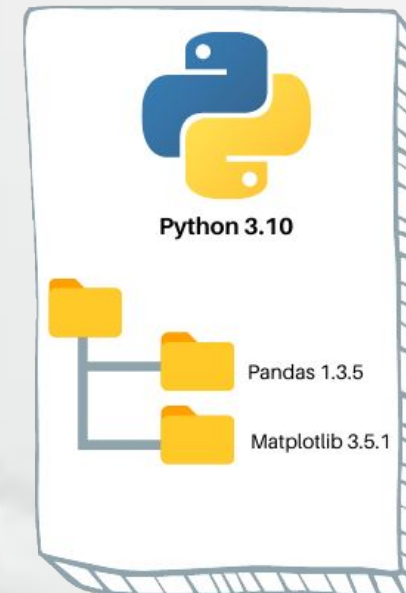
Virtual Environment 1



Virtual Environment 2



Virtual Environment 3



Start PowerShell
[wt.bat](#) WSL Linux

Acceso al [IDLE](#)



PYTHON

TIPOS DE DATOS

INTRODUCCION AL OS TIPOS DE DATOS EN PYTHON.

Tipos de dato

Los lenguajes de programación nos permiten almacenar datos en la memoria del sistema, y ello a través de **variables**. “Pero una de las diferencias fundamentales entre un lenguaje u otro es el modo en que maneja esas variables”. De algunos decimos que son **estáticos (C++,Java,C#)**, por cuanto una caja solamente puede almacenar datos de un tipo en particular; mientras que de otros se dice que son **dinámicos (Smaltalk)**, cuando una misma variable puede contener un número al comienzo del programa y luego albergar una cadena de caracteres, por ejemplo.

Según esta clasificación, diremos que **Python es un lenguaje dinámico**. No obstante, la forma en la que entiende las “**variables**” es un tanto diferente respecto de otros lenguajes. En lugar de concebirlas como “cajas”, las entiende como nombres o *referencias*. Por ello, en lugar de “variable” generalmente se emplea el término objeto (aunque bien las utilizaremos de forma indistinta).

Tipos básicos

Hechas las presentaciones, ya es hora de crear nuestro primer objeto. “**Python no distingue la creación de la asignación**”.

Un objeto es creado automáticamente cuando le asignamos un valor.

Así, para crear el objeto a con el valor 1 simplemente haremos lo siguiente.

```
>>> a = 1
```

¡Perfecto! Y dado que la creación ocurre de forma implícita en la asignación, tampoco es necesario indicar el tipo de dato que contendrá el objeto pues es *inferido por Python*.

En la consola interactiva, podemos escribir el nombre de un objeto para ver su valor.

```
>>> a
1
```

Para conocer de qué tipo es un objeto, empleamos la función incorporada (recuerda, aquellas que están definidas por Python) `type()`.

```
>>> type(a)
<class 'int'>
```

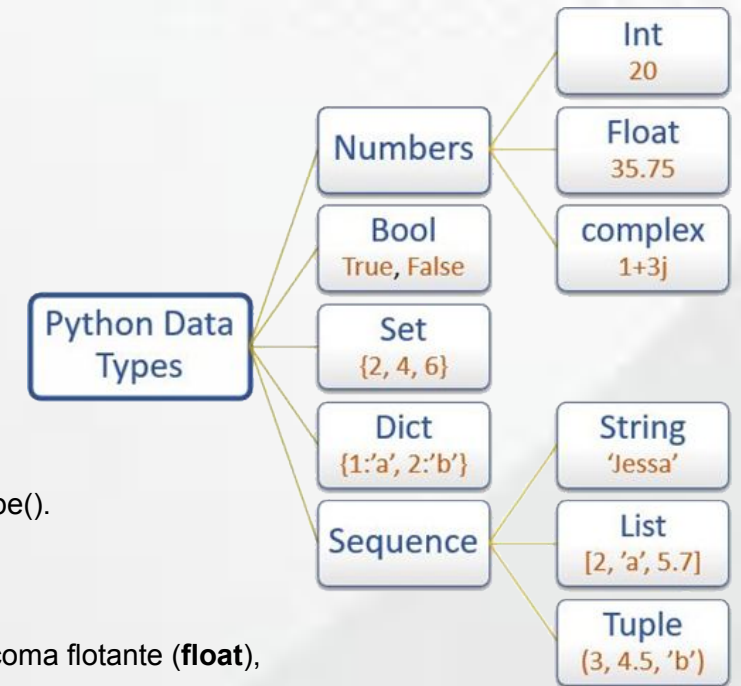
Como vemos, nuestro objeto es del tipo **int**: esto es, un número entero. Los otros tres tipos de dato básicos son los números de coma flotante (**float**), los booleanos (**bool**) y las cadenas de caracteres (**str**).

```
>>> pi = 3.14
>>> type(pi)
<class 'float'>
>>> prendido = True
>>> type(prendido)
<class 'bool'>
```

```
>>> s = "¡Hola, mundo!"
>>> type(s)
<class 'str'>
```

Nótese que los nombres de las variables se escriben en letras minúsculas y separadas por guiones bajos.

Los valores posibles para un booleano son True y False. Estas dos palabras están reservadas por el lenguaje, lo que quiere decir que no podemos definir objetos con esos nombres.



Language Tips: Python tiene una peculiaridad y trata sus tipos de datos como mutables o inmutables, lo que significa que si el valor puede cambiar, el objeto se llama mutable, mientras que si el valor no puede cambiar, el objeto se llama inmutable.

Acceso al [IDLE](#)



Operaciones aritméticas y de comparación

Las operaciones aritméticas son la suma, la resta, la multiplicación y la división. Los operadores utilizados son similares a los de otros lenguajes de programación.

```
>>> a = 5
>>> b = 7
>>> a + b
12
>>> a - b
-2
>>> a * b
35
>>> a / b
0.7142857142857143
```

Todas estas operaciones son expresiones, es decir, sentencias de código que devuelven un resultado. Cuando escribimos una expresión en la consola interactiva, lo que resulta de ello es impreso en la pantalla.

A estas cuatro podemos añadir la división integral y el resto.

División integral: siempre retorna un número entero.

```
>>> 10 // 3
3
```

Resto o módulo: retorna el remanente de la división 16 / 5.

```
>>> 16 % 5
1
```

En Python, un script es un archivo de texto plano que contiene una serie de instrucciones que se ejecutan en orden secuencial. Estas instrucciones son escritas en lenguaje Python y pueden incluir variables, operadores, estructuras de control de flujo, funciones y clases. Los scripts en Python se utilizan comúnmente para automatizar tareas repetitivas o para procesar grandes cantidades de datos. Por ejemplo, un script en Python podría leer un archivo de datos, realizar algún tipo de análisis o procesamiento en los datos y luego escribir los resultados en otro archivo.

Ahora bien, en cuanto a comparación de números se refiere, contamos con los operadores > (mayor), >= (mayor o igual), < (menor) y <= (menor o igual). Las operaciones de comparación siempre retornan un booleano.

```
>>> 10 > 5      >>> 6 >= 5
True           True
>>> 10 < 5      >>> 4 <= 2
False          False
```



Para saber si el valor de dos objetos es igual o distinto, empleamos los operadores == y !=.

```
>>> 5 == 2      >>> "Hola mundo!" != 3.14
False           True
```

Como habrás observado, no es necesario que los objetos que conforman una comparación sean del mismo tipo de dato.

Arithmetic Operators			<div>a = 5 b = 2</div>
Operator	Meaning	Example	Result
+	Addition Operator. Adds two Values.	a + b	7
-	Subtraction Operator. Subtracts one value from another	a - b	3
*	Multiplication Operator. Multiplies values on either side of the operator	a * b	10
/	Division Operator. Divides left operand by the right operand.	a / b	2.5
%	Modulus Operator. Gives reminder of division	a % b	1
**	Exponent Operator. Calculates exponential power value. a ** b gives the value of a to the power of b	a ** b	25
//	Integer division, also called floor division. Performs division and gives only integer quotient.	a // b	2



PYTHON

CONTROL DE FLUJO

CONTROL DE FLUJO Y BUCLES EN PYTHON.

Control de flujo

Un programa o **script** de Python es un conjunto de instrucciones analizadas y ejecutadas por el intérprete de arriba hacia abajo y de izquierda a derecha. Cuando todas las instrucciones se han ejecutado, el programa termina. No obstante, contamos con herramientas para alterar el flujo natural del programa: hacer que se saltee una porción de código según se cumpla tal o cual condición, repetir un conjunto de instrucciones, etc.

Condicional

Una de estas herramientas es el condicional. A partir de la palabra reservada `if`, le indicamos a Python que queremos ejecutar una porción de código solo si se cumple una determinada condición (es decir, si el resultado es `True`, como vimos anteriormente). Consideremos el siguiente ejemplo.

```
edad = 30
```

```
if edad >= 18:  
    print("Eres un adulto.")
```

Primero definimos una variable `edad`, cuyo valor es un entero, 30. Luego, a través del condicional indicamos que queremos imprimir "Eres un adulto." en pantalla si se cumple la condición de que el valor de `edad` sea mayor o igual a 18.

En Python se emplean cuatro espacios en blanco (4) para delimitar los bloques de código.

```
if edad >= 18:  
    print("Eres un adulto.")  
print("¡Hola, mundo!")
```

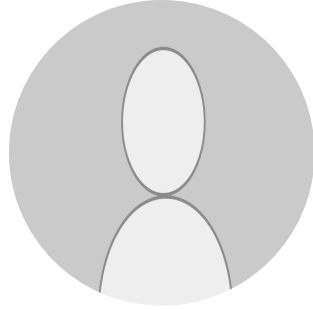
En este código, solamente la segunda línea está dentro del condicional, por cuanto está "indentada" con cuatro espacios. Por el contrario, la tercera línea se ejecuta siempre, independientemente del resultado de la comparación.



Un script es un archivo de texto que contiene un conjunto de comandos o instrucciones que son ejecutados por un programa o sistema operativo en lugar de ser ejecutados manualmente por un usuario.

Los scripts pueden ser utilizados para automatizar tareas repetitivas, como la instalación de software, la configuración de sistemas, la realización de copias de seguridad, la generación de informes, entre otras. Los scripts pueden ser escritos en diferentes lenguajes de programación, como Bash, Python, Ruby, Perl, entre otros.

Control de flujo



Control de flujo

Ahora bien, para ejecutar un bloque de código en caso que no se cumpla la condición, empleamos la palabra reservada **else**.

```
if edad >= 18:  
    print("Eres un adulto.")  
else:  
    print("Aún no eres un adulto.")
```

Por último, podemos especificar otras condiciones en caso que la primera no se cumpla vía **elif**.

```
if edad >= 18 and edad < 65:  
    print("Eres un adulto.")  
elif edad >= 65:  
    print("Eres un adulto mayor.")  
else:  
    print("Aún no eres un adulto.")
```

if()

Aquí, además, hemos incluido una conjunción vía la palabra reservada **and** para indicar que, en el primer caso, para que se cumpla la condición, tanto `edad >= 18` como `edad < 65` deben ser verdaderos. Lo contrario de la conjunción es la disyunción, representada por la palabra reservada **or**.

Interacción con el usuario



Interacción con el Usuario x Consola

Python incluye una función llamada **input()**, similar a **print()**, pero que exhorta al usuario a escribir algo en la consola, que luego es retornado como una cadena.

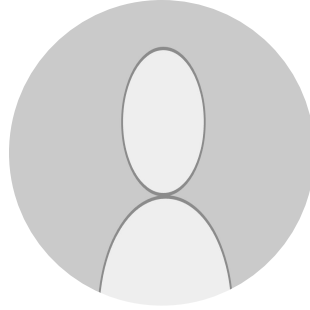
```
nombre = input("Escribe tu nombre: ")  
print("Hola", nombre)
```

(La función **print()** nos permite imprimir múltiples objetos en una misma línea separándolos por comas, que luego se traducen en la pantalla como espacios).

Así, este código solicita al usuario que escriba su nombre y, una vez obtenido, imprime un saludo personalizado en la pantalla. Sabido esto, podemos adaptar nuestro código anterior para solicitar la edad del usuario y en consecuencia mostrar el mensaje correspondiente.

```
edad = int(input("Escribe tu edad: "))  
  
if edad >= 18 and edad < 65:  
    print("Eres un adulto.")  
elif edad >= 65:  
    print("Eres un adulto mayor.")  
else:  
    print("Aún no eres un adulto.")
```

Dado que **input()** siempre retorna una cadena de caracteres, debemos primero convertirla a un entero vía la función incorporada **int()** para poder efectuar las comparaciones pertinentes.



input()





PYTHON

COLECCIONES

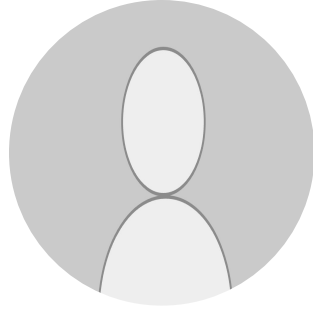
INTRODUCCION A LAS COLECCIONES EN
PYTHON.
En Python existen cuatro colecciones básicas:

listas
tuplas
diccionarios
set

Colecciones

LISTAS

Una colección permite agrupar varios objetos bajo un mismo nombre. Por ejemplo, si necesitamos almacenar en nuestro programa los nombres de los alumnos de un curso de programación, será más conveniente ubicarlos a todos dentro de una misma colección de nombre alumnos, en lugar de crear los objetos alumno1, alumno2, etc.



Listas

Una lista es un conjunto ordenado de objetos. Por objetos entendemos cualquiera de los tipos de dato ya mencionados, incluso otras listas.

Para crear una lista, especificamos sus elementos entre corchetes y separados por comas.

```
>>> lenguajes = ["Python", "Java", "C", "C++"]
```

Dado que se trata de una colección ordenada, accedemos a cada uno de los elementos a partir de su posición, comenzando desde el 0.

```
>>> lenguajes[0]
```

```
'Python'
```

```
>>> lenguajes[1]
```

```
'Java'
```

(En Python, las cadenas de caracteres pueden estar formadas tanto por comillas dobles como por comillas simples).

La posición de un elemento, también conocida como índice, aumenta de izquierda a derecha cuando éste es positivo. Indicando un número negativo, los elementos se obtienen de derecha a izquierda.

```
>>> lenguajes[-1]
```

```
'C++'
```

```
>>> lenguajes[-2]
```

```
'C'
```

Language Tips: Python does not have built-in support for Arrays, but [Python Lists](#) can be used instead.

Colecciones

LISTAS

Una colección permite agrupar varios objetos bajo un mismo nombre. Por ejemplo, si necesitamos almacenar en nuestro programa los nombres de los alumnos de un curso de programación, será más conveniente ubicarlos a todos dentro de una misma colección de nombre alumnos, en lugar de crear los objetos alumno1, alumno2, etc.

Listas – Operaciones (Modificar – Eliminar – Insertar)

MODIFICAR: Para **cambiar** un elemento, combinamos la nomenclatura descrita con la asignación.

```
>>> lenguajes[1] = "Rust"
>>> lenguajes
['Python', 'Rust', 'C', 'C++']
```

ELIMINAR: Para **remove** un elemento, utilizamos la palabra reservada **del**.

```
>>> del lenguajes[1]
>>> lenguajes
['Python', 'C', 'C++']
```

Nótese que cuando un elemento es eliminado, a excepción del último, todos los que lo suceden se corren una posición hacia la izquierda.

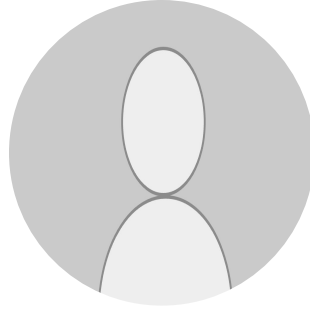
INSERTAR: La operación inversa es la de **insertar** un elemento en una posición determinada, desplazando el resto que le sucede una posición hacia la derecha, vía el método (volveremos sobre este concepto más adelante) `insert()`.

Insertar la cadena "Elixir" en la posición 1.

```
>>> lenguajes.insert(1, "Elixir")
>>> lenguajes
['Python', 'Elixir', 'C', 'C++']
```

INSERTAR AL FINAL: Por último, la operación más común es la de **agregar un elemento** al final de la lista. Para ello empleamos `append()`.

```
>>> lenguajes.append("Haskell")
>>> lenguajes
['Python', 'Elixir', 'C', 'C++', 'Haskell']
```



Colecciones

TUPLAS

Las tuplas, al igual que las listas, son colecciones ordenadas. No obstante, a diferencia de éstas, son **inmutables**. Es decir, una vez asignados los elementos, no pueden ser alterados. En términos funcionales, podría decirse que las tuplas son un subconjunto de las listas, por cuanto soportan las operaciones con índices para acceder a sus elementos, pero no así las de asignación.

Tuplas en Python

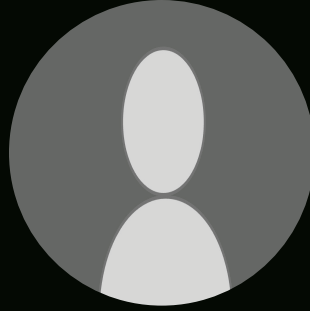
```
# Indicamos los elementos entre paréntesis.  
>>> lenguajes = ("Python", "Java", "C", "C++")  
>>> lenguajes[0]  
'Python'  
>>> lenguajes[-1]  
'C++'
```

Un intento por remover un elemento de una tupla o asignarle un nuevo valor arrojará un error.

Ahora bien, si las listas hacen todo lo que hace una tupla, ¿para qué, entonces, existen las tuplas?

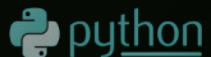
Sencillamente porque una tupla, justamente por carecer de todas las funcionalidades de mutación de elementos, es un objeto más liviano y que se crea más rápido.

La función del programador será la de dirimir si para una determinada colección ordenada de objetos le conviene más una lista o una tupla, según sus elementos deban ser alterados en el futuro o no.



Language Tips
Python tiene una peculiaridad y trata sus tipos de datos como mutables o inmutables, lo que significa que si el valor puede cambiar, el objeto se llama mutable, mientras que si el valor no puede cambiar, el objeto se llama inmutable.

Opcional: ¿Cómo funcionan? ¿Qué son? Mutables e Inmutables - Jarroba



Colecciones

DICCIONARIOS

Los diccionarios, a diferencia de las listas y las tuplas, son colecciones no ordenadas de objetos. Además, sus elementos tienen una particularidad: siempre conforman un par clave-valor. Es decir, cuando añadimos un valor a un diccionario, se le asigna una clave única con la que luego se podrá acceder a él (pues la posición ya no es un determinante).

Diccionarios en Python

Para crear un diccionario, indicamos los pares **clave-valor** separados por comas y estos, a su vez, separados por dos puntos.

```
>>> d = {"Python": 1991, "C": 1972, "Java": 1996}
```

En este ejemplo, creamos un diccionario con tres elementos, cuyas claves son "Python", "C" y "Java" y sus valores los años en los que fueron creados, a saber: 1991, 1972 y 1996, respectivamente.

Para acceder a cualquiera de los valores, debemos indicar su clave entre corchetes.

```
>>> d["Java"]
1996
```

Las operaciones **de añadir, modificar o eliminar** valores son similares a las de las listas.

Añadir un par clave/valor.

```
>>> d["Elixir"] = 2011
```

Modificar uno existente.

```
>>> d["Python"] = 3000
```

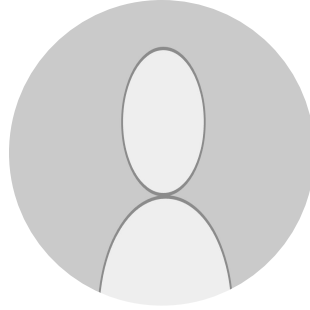
Eliminar.

```
>>> del d["C"]
```

Mostrar

```
>>> d
{'Python': 3000, 'Java': 1996, 'Elixir': 2011}
```

Las claves típicamente serán números o cadenas, aunque bien podrían ser de cualquier otro tipo de dato inmutable (todos los vistos hasta ahora son inmutables, a excepción de las listas y los diccionarios). Además, nunca puede haber claves repetidas, ya que actúan como identificadores únicos para un valor determinado. Los valores, en cambio, pueden ser de cualquier tipo de dato, ¡incluyendo otros diccionarios!



Operaciones comunes

DICCIONARIOS

Hemos visto cómo se aplica tanto a listas como a diccionarios. Existen varias funciones que operan sobre distintos tipos de colecciones. Por ejemplo, la función incorporada `len()` retorna el número de elementos en una colección

En la tercera llamada pasamos como argumento una tupla que contiene un solo element (una cadena). No obstante, dado que para definir tuplas se emplean paréntesis, es necesario añadir una coma al final para que Python entienda que se trata de una tupla en lugar de una expresión (los paréntesis también son utilizados para agrupar expresiones, como en matemática).

```
>>> len([1, 2, 3, 4])
4
>>> len({"Python": 1991, "C": 1972})
2
>>> len(("¡Hola, mundo!"))
1
```

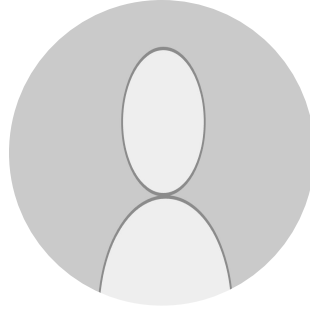
Una cadena es, en rigor, una colección de caracteres. Por ello también soporta las operaciones de acceso a sus elementos (¡no así las de asignación, ya que son inmutables!).

```
>>> s = "¡Hola, mundo!"
>>> s[4]
'a'
>>> len(s)
13
```

Por último, para saber si un elemento está dentro de una colección, empleamos la palabra reservada `in`.

```
>>> 10 in (1, 2, 3, 4)
False
>>> "mundo" in "¡Hola, mundo!"
True
>>> "Python" in {"Python": 1991, "C": 1972}
True
```

Nótese que el operador `in` en los diccionarios actúa sobre las claves, no sobre los valores.



Colecciones

SET

set en Python es utilizado para trabajar con conjuntos de elementos. La principal característica de este tipo de datos es que es una colección cuyos elementos no guardan ningún orden y que además son únicos.

Estas características hacen que los principales usos de esta clase sean conocer si un elemento pertenece o no a una colección y eliminar duplicados de un tipo secuencial (list, tuple o str)..

¿Cómo funcionan? ¿Qué son? Mutables e Inmutables - Jarroba

Sets en Python

En Python, un conjunto (set en inglés) es una colección desordenada de elementos únicos. Los elementos en un conjunto no están en ningún orden específico y no se pueden acceder por índice. En cambio, se puede verificar si un elemento está en un conjunto utilizando el operador in.

Aquí hay un ejemplo de cómo crear un conjunto en Python:

```
mi_set = {'banana', 'manzana', 'cereza' }  
print(mi_set)
```

Esto imprimirá: {'banana', 'manzana', 'cereza'}. Como puedes ver, los elementos no están en ningún orden específico.

También puedes crear un conjunto vacío usando set():

```
mi_set_vacio = set()
```

Puedes agregar elementos a un conjunto usando el método add():

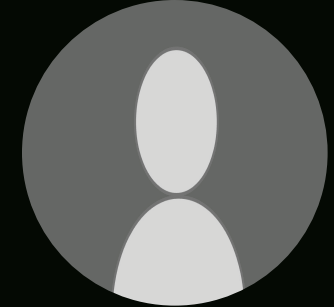
```
mi_set.add("dátil")  
print(mi_set)
```

Esto imprimirá: {'banana', 'manzana', 'cereza', 'dátil'}.

Comprobar la existencia

Podemos comprobar si un elemento existe en la colección con la palabra clave in. Devolverá True o False dependiendo de si existe o no.

```
frutas = {'platano', 'manzana', 'naranja'}  
print('platano' in frutas) print('coco' in frutas)  
print('coco' not in frutas)
```



Language *Tips*

Python tiene una peculiaridad y trata sus tipos de datos como mutables o inmutables, lo que significa que si el valor puede cambiar, el objeto se llama mutable, mientras que si el valor no puede cambiar, el objeto se llama inmutable.





PYTHON

BUCLES

INTRODUCCION A LAS BUCLES EN PYTHON.

Los bucles son otra herramienta para alterar el flujo normal de un programa. Nos permiten repetir una porción de código tantas veces como queramos. Python incluye únicamente dos tipos de bucle: `while` y `for`.

While
For
Range
Switch



Bucle «while»

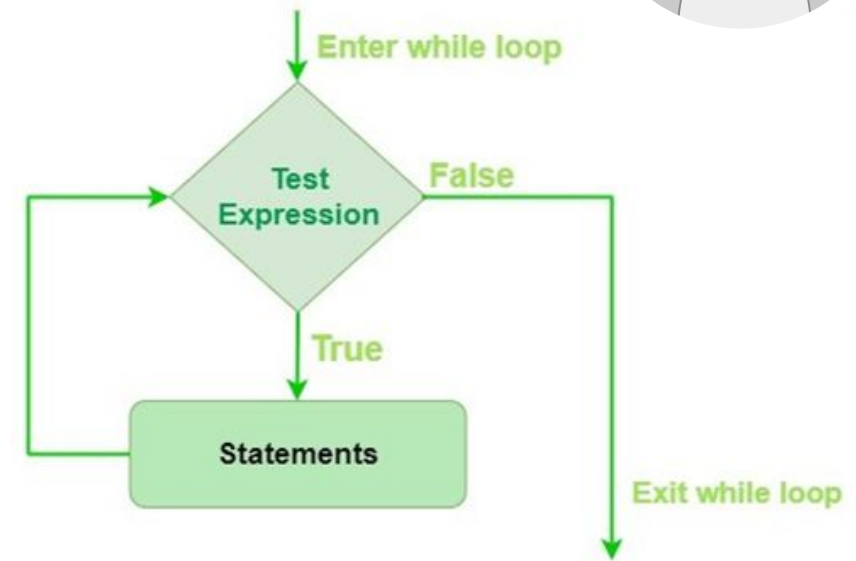
La palabra reservada `while` ejecuta una porción de código una y otra vez hasta que la condición especificada sea falsa; o, dicho de otro modo, ejecuta una porción de código mientras que la condición sea verdadera.

```
a = 1
while a < 10:
    print("¡Hola, mundo!")
```

Si ejecutamos este código, veremos que el programa imprime infinitamente el mensaje en pantalla. Esto ocurre por cuanto la condición `a < 10` es inexorablemente siempre verdadera (uno es menor que diez). Ahora bien, consideremos este otro ejemplo.

```
a = 1
while a < 10:
    print("¡Hola, mundo!")
    a = a + 1
```

Ahora el resultado es distinto: el mensaje se imprime nueve veces. Analicemos por qué pasa esto. Cuando el intérprete tiene que ejecutar el bucle, evalúa la condición `a < 10`, que en un primer momento equivale a `1 < 10`, que es verdadero. Entonces, ejecuta el bloque de código dentro del bucle: imprime el mensaje en pantalla y luego le suma 1 a la variable `a`. Cuando el bloque de código finaliza, se vuelve a evaluar la condición, que ahora resulta ser `2 < 10`, aún verdadero. Luego se ejecuta nuevamente el bloque de código y el proceso se repite hasta llegar a la condición `10 < 10`, que es falsa y por ende el bucle termina.



Bucles While



Bucle «while»

Podemos ver con más detalle el proceso imprimiendo la variable a.

```
a = 1
while a < 10:
    print("¡Hola, mundo!")
    print(a)
    a += 1
```

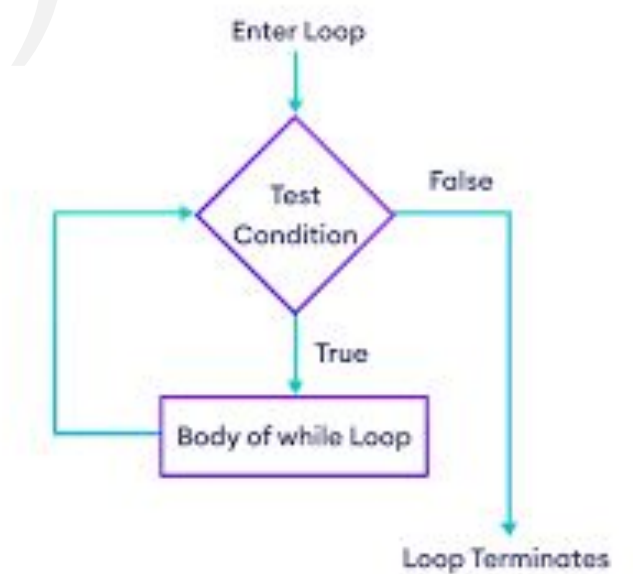
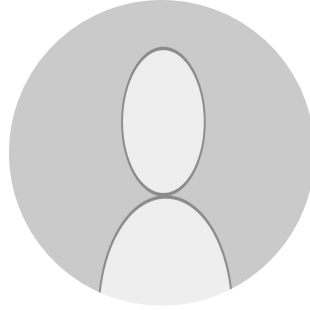
Además, cambié la última línea por `a += 1`, que es simplemente un atajo para `a = a + 1`.

Ahora bien, combinando lo que hemos aprendido hasta el momento y el conocimiento que obtuvimos sobre colecciones en la sección anterior, podemos crear un bucle while que imprima todos los elementos dentro de una list:

```
lenguajes = ["Python", "C", "C++", "Java"]
i = 0
while i < len(lenguajes):
    print(lenguajes[i])
    i += 1
```

¡Excelente! No importa cuántos elementos agregemos a `lenguajes`, nuestro bucle estará preparado para imprimir todos ellos.

while()



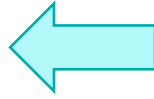
Bucles While



Bucle «for»

Si bien un bucle while es perfectamente capaz de, como vimos, imprimir los elementos de una colección, resulta que un bucle for es una herramienta más pertinente para este tipo de situaciones.

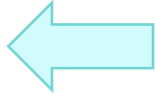
```
lenguajes = ["Python", "C", "C++", "Java"]  
for lenguaje in lenguajes:  
    print(lenguaje)
```



¡Genial! Lo que ha ocurrido aquí es lo siguiente: el bucle for ejecuta el bloque de código indentado (en este caso la llamada a print()) tantas veces como elementos haya en la colección indicada a la derecha del operador in. Pero, cada vez que ese código es ejecutado, la variable lenguaje tendrá un valor diferente: en la primera ejecución será igual a "Python"; en la segunda, a "C"; y así hasta alcanzar el final de la lista.

Este tipo de bucle for es bastante diferente al que prima en otros lenguajes de programación. No obstante, ese comportamiento puede conseguirse fácilmente en Python empleando otra de sus funciones incorporadas. Por ejemplo, si queremos imprimir números del 1 al 10:

```
for i in range(1, 11):  
    print(i)
```

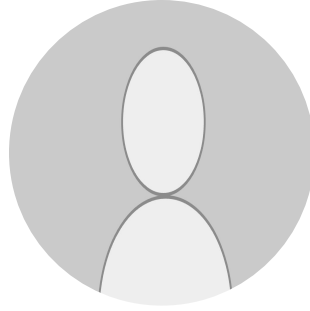


Te lo cuento en la
Sigüiente Diapositiva..

¿Qué tipos de dato pueden ser recorridos con un bucle for? Bien, de los que hemos visto, las colecciones (listas, tuplas y diccionarios) y las cadenas (entendidas como una colección de caracteres). No obstante, hay muchos otros objetos que no se incluyen en ninguna de estas categorías y no obstante ello pueden ser recorridos. Se dice que son objetos iterables. Este es el caso del objeto retornado por la función range().

Históricamente, range() retornaba una lista, de modo que range(1, 4) == [1, 2, 3]. En versiones actuales de Python, su funcionamiento es un tanto más complejo, por cuanto genera los números del rango especificado a medida que son utilizados, para optimizar el rendimiento del código. No te preocupes, no es una cuestión relevante en este momento. En última instancia, puedes convertir cualquier objeto iterable a una lista vía list().

```
>>> list(range(1, 11))  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```



Bucles For



Uso del range

```
for i in (0, 1, 2, 3, 4, 5):  
    print(i)    #0, 1, 2, 3, 4, 5
```

Se trata de una solución que cumple con nuestro requisito. El contenido después del in se trata de una clase que como ya hemos visto antes, es iterable, y es de hecho una tupla. Sin embargo, hay otras formas de hacer esto en Python, haciendo uso del range().

```
for i in range(6):  
    print(i)    #0, 1, 2, 3, 4, 5
```

El range() genera una secuencia de números que van desde 0 por defecto hasta el número que se pasa como parámetro menos 1. En realidad, se pueden pasar hasta tres parámetros separados por coma, donde el primer es el inicio de la secuencia, el segundo el final y el tercero el salto que se desea entre números. Por defecto se empieza en 0 y el salto es de 1.

```
#range(inicio, fin, salto)
```

Por lo tanto, si llamamos a range() con (5,20,2), se generarán números de 5 a 20 de dos en dos. Un truco es que el range() se puede convertir en list.

```
print(list(range(5, 20, 2)))
```

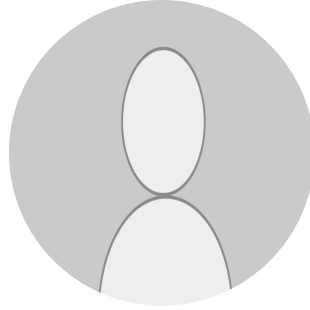
Y mezclándolo con el for, podemos hacer lo siguiente.

```
for i in range(5, 20, 2):  
    print(i) #5,7,9,11,13,15,17,19
```

Una de las iteraciones mas comunes que se realizan, es la de iterar un número entre por ejemplo 0 y n. En c++ lo vimos. Pongamos que queremos iterar una variable i de 0 a 5. Haciendo uso de lo que hemos visto anteriormente puedes usar Range.

Se pueden generar también secuencias inversas, empezando por un número mayor y terminando en uno menor, pero para ello el salto deberá ser negativo.

```
for i in range (5, 0, -1):  
    print(i) #5,4,3,2,1
```



Range en Python



Switch en Python

El switch es una herramienta que nos permite ejecutar diferentes secciones de código dependiendo de una condición. Su funcionalidad es similar a usar varios if, pero por desgracia Python no tiene un switch propiamente dicho. Sin embargo, hay formas de simular su comportamiento que te explicamos a continuación.

Introducción al switch

Ya sabemos que el uso del if junto con else y elif nos permite ejecutar un Código determinado dependiendo de una condición, como podemos ver en el siguiente código.

```
if condicion == 1:  
    print("Haz a")  
elif condicion == 2:  
    print("Haz b")  
elif condicion == 3:  
    print("Haz c")  
else:  
    print("Haz d")
```

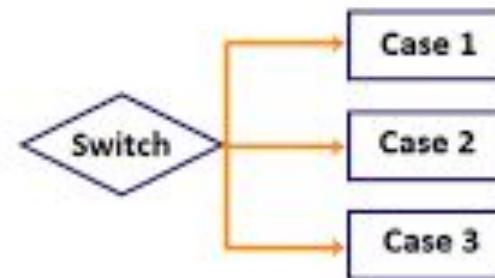
La misma funcionalidad se podría escribir de la siguiente manera haciendo uso del switch. Como puedes ver su uso tal vez resulte algo más limpio, y de hecho en determinadas ocasiones es más rápido.

// Ojo, esto no es Python

```
switch(condicion) {  
    case 1:  
        // haz a  
        break;  
    case 2:  
        // haz b  
        break;  
    case 3:  
        // haz c  
        break;  
    default:  
        // haz x  
}
```

X-Switch

SWITCH CASE IN PYTHON



Switch en Python

El switch es una herramienta que nos permite ejecutar diferentes secciones de código dependiendo de una condición. Su funcionalidad es similar a usar varios if, pero por desgracia Python no tiene un switch propiamente dicho. Sin embargo, hay formas de simular su comportamiento que te explicamos a continuación.

Pero tenemos un pequeño problema. En Python no existe el switch, por lo que si intentas ejecutar el Código anterior, tendrás un error.

Uno tal vez podría decir: bueno, que más da, uso if/elif/else y ya está. La verdad que en la mayoría de los casos, sería indiferente usar if o switch, pero si analizamos el comportamiento que existe por debajo, funcionan de manera distinta. A pesar de que en Python no existe, veremos un truco que puede en cierto modo emular su funcionamiento.

Diferencia if y switch

Una de las principales diferencias, es que usando if con elif, no todos los bloques tienen el mismo tiempo de acceso. Todas las condiciones van siendo evaluadas hasta que se cumple y se sale. Imaginemos que tenemos 100 condiciones.

```
if condicion == 1:
    print("1")
elif condicion == 2:
    print("2")
# ... hasta 100
elif condicion == 100:
    print("3")
else:
    print("x")
```

El tiempo de ejecución será distinto si la condición es 1 o es 70 por ejemplo:
(Veremos Complejidad Algorítmica en la Unidad 1 en 3ro, con Notación Asintótica)

Si es 1: Se evalúa el primer if, y como se cumple la condición se ejecuta y se sale.

Si es 70: Se va evaluando cada condición, hasta llegar al 70. Es decir, tienen que evaluarse 70 condiciones.

Sin embargo, en el switch todos los elementos tienen el mismo tiempo de acceso. Esto se debe a que por debajo es normalmente implementado con lookup tables.

Si trabajamos con un gran número de condiciones, el uso del switch sobre el if podría notarse.

Switch en Python

Dado que en Python no tenemos esta herramienta, te explicamos un truco para simularlo. No obstante, si realmente te preocupan estas micro optimizaciones, tal vez no deberías usar Python.

Emulando switch en Python

Una forma de tener una especie de switch en Python es haciendo uso de un diccionario. Por lo tanto podríamos convertir el siguiente código.

```
def opera1(operador, a, b):
```

```
    if operador == 'suma':
```

```
        return a + b
```

```
    elif operador == 'resta':
```

```
        return a - b
```

```
    elif operador == 'multiplica':
```

```
        return a * b
```

```
    elif operador == 'divide':
```

```
        return a / b
```

```
    else:
```

```
        return None
```

`opera1('suma', 5, 9) # Salida: 14`

O el siguiente Usando Lambda:

```
def opera2(operador, a, b):
```

```
    return {
```

```
        'suma': lambda: a + b,
```

```
        'resta': lambda: a - b,
```

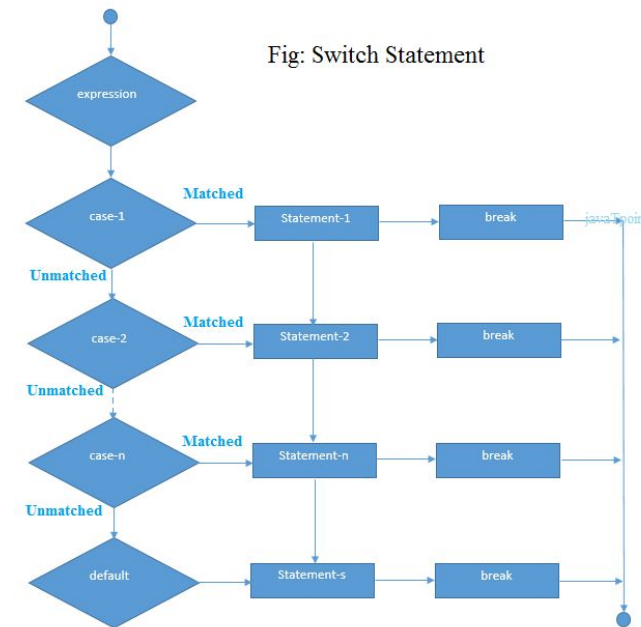
```
        'multiplica': lambda: a * b,
```

```
        'divide': lambda: a / b
```

```
    }.get(operador, lambda: None)
```

`opera2('suma', 5, 9)() # Salida: 14`

Nota: Esto es Programación Funcional en Python. Lo veremos mas adelante...



Sentencia break Python

La sentencia **break** nos permite alterar el comportamiento de los bucles while y for. Concretamente, permite terminar con la ejecución del bucle.

Esto significa que una vez se encuentra la palabra break, el bucle se habrá terminado.

Break con bucles for

Veamos como podemos usar el break con bucles for. El range(5) generaría 5 iteraciones, donde la i valdría de 0 a 4. Sin embargo, en la primera iteración, terminamos el bucle prematuramente.

El break hace que nada más empezar el bucle, se rompa y se salga sin haber hecho nada.

```
for i in range(5):
```

```
    print(i)
```

```
    break
```

```
    # No llega
```

```
# Salida: 0
```

Un ejemplo un poco más útil, sería el de buscar una letra en una palabra. Se itera toda la palabra y en el momento en el que se encuentra la letra que buscábamos, se rompe el bucle y se sale.

Esto es algo muy útil porque si ya encontramos lo que estábamos buscando, no tendría mucho sentido seguir iterando la lista, ya que desperdiciaríamos recursos.

```
cadena = 'Python'
```

```
for letra in cadena:
```

```
    if letra == 'h':
```

```
        print("Se encontró la h")
```

```
        break
```

```
    print(letra)
```

```
# Salida:
```

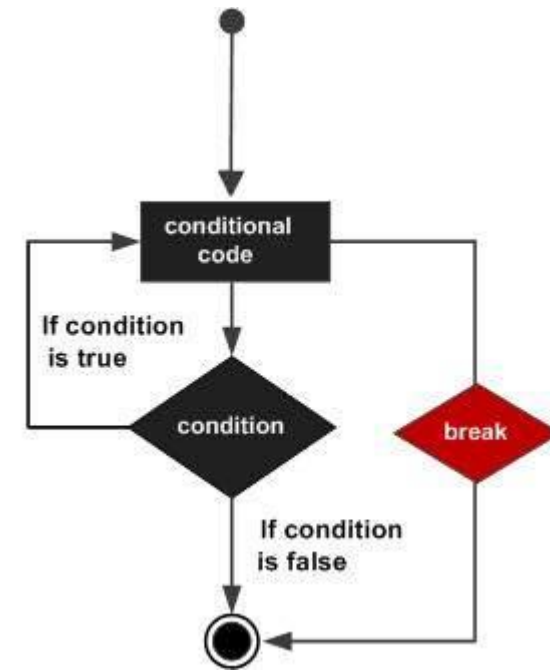
```
# P
```

```
# y
```

```
# t
```

```
# Se encontró la h
```

```
Break con bucles while
```



Sentencia break Python

El break también nos permite alterar el comportamiento del while. Veamos un ejemplo.

La condición while True haría que la sección de código se ejecutara indefinidamente, pero al hacer uso del break, el bucle se romperá cuando x valga cero.

```
x = 5
while True:
    x -= 1
    print(x)
    if x == 0:
        break
    print("Fin del bucle")
```

#4, 3, 2, 1, 0

Por norma general, y salvo casos muy concretos, si ves un while True, es probable que haya un break dentro del bucle.

Break y bucles anidados

Como hemos dicho, el uso de break rompe el bucle, pero sólo aquel en el que está dentro.

Es decir, si tenemos dos bucles anidados, el break romperá el bucle anidado, pero no el exterior.

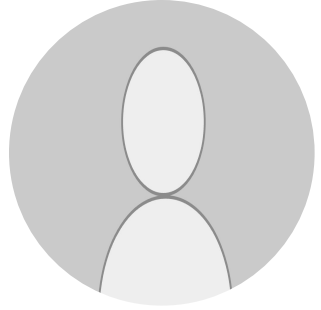
```
for i in range(0, 4):
    for j in range(0, 4):
        break
    #Nunca se realiza más de una iteración
    # El break no afecta a este for
    print(i, j)
```

0 0

1 0

2 0

3 0



break

Sentencia continue

El uso de continue al igual que el ya visto break, nos permite modificar el comportamiento de los bucles while y for.

Concretamente, continue se salta todo el código restante en la iteración actual y vuelve al principio en el caso de que aún queden iteraciones por completar.

La diferencia entre el break y continue es que el continue no rompe el bucle, si no que pasa a la siguiente iteración saltando el código pendiente

En el siguiente ejemplo vemos como al encontrar la letra P se llama al continue, lo que hace que se salte el print(). Es por ello por lo que no vemos la letra P impresa en pantalla.

```
cadena = 'Python'
for letra in cadena:
    if letra == 'P':
        continue
    print(letra)
```

Salida:

```
# y
# t
# h
# o
# n
```

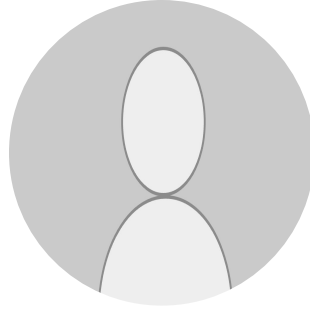
A diferencia del break, el continue no rompe el bucle sino que finaliza la iteración actual, haciendo que todo el código que va después se salte, y se vuelva al principio a evaluar la condición.

En el siguiente ejemplo podemos ver como cuando la x vale 3, se llama al continue, lo que hace que se salte el resto de código de la iteración (el print()). Por ello, vemos como el número 3 no se imprime en pantalla.

```
x = 5
while x > 0:
    x -= 1
    if x == 3:
        continue
    print(x)
```

#Salida: 4, 2, 1, 0

continue



7 Py





PYTHON

FUNCIONES

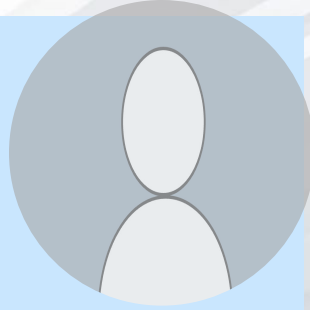
INTRODUCCION A LAS FUNCIONES EN PYTHON.



Funciones

Hay dos tipos de Funciones:

- El concepto de función tal como se conoce en matemática: una función obtiene un valor o un conjunto de valores de entrada, aplica una serie de operaciones y retorna un valor o un conjunto de valores de salida, un resultado.
- Otra, una función que puede categorizarse junto con los condicionales y los bucles: esto es, sirve para agrupar un conjunto de instrucciones bajo un mismo nombre para evitar repetirlas a lo largo del código.



Definamos una función `dup(n)`, muy al estilo matemático, que tome como argumento un número y retorne el doble.

```
def dup(n):  
    return n * 2
```

Introducimos dos palabras reservadas nuevas. **def** es empleada siempre que se quiera crear una nueva función, seguida de su nombre (que, al igual que las variables, se escribe en minúscula y separada por guiones bajos) y los argumentos entre paréntesis y separados por comas. **return** debe estar sucedida por una expresión que será el valor de retorno de la función. Iremos aclarando estos conceptos.

Por el momento, hagamos una prueba:

```
# Imprime 10 en pantalla.  
print(dup(5))
```

Ahora consideremos esta otra definición.

```
def saludar(nombre):  
    print("Hola", nombre)
```

```
saludar("mundo")
```

En este caso, nuestra función `saludar()` no retorna ningún valor. Aunque, en efecto, Python le asigna un valor de retorno por defecto llamado `None`. Se trata de un tipo de dato como los que hemos estado trabajando, pero un tanto más particular, ya que intenta indicar que una variable está vacía.

```
>>> a = None  
>>> b = 3.14  
>>> a is None  
True  
>>> b is not None  
True
```

Nota: La única diferencia, como se observa, es que para realizar comparaciones respecto de **None** utilizamos las palabras reservadas **is** e **is not** en lugar de `==` y `!=`.



Argumentos en Funciones



En Python los argumentos de las funciones tienen un comportamiento particular a diferencia de otros lenguajes. Tomemos como ejemplo la siguiente función.

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

De ella decimos que tiene dos **argumentos posicionales**. Llevan ese nombre por cuanto al invocar `sumar(7, 5)`, 7 se corresponde con `a` por ser el primer argumento, y 5 se corresponde con `b` por ser el segundo. Si lo invertimos, de modo que llamemos a `sumar(5, 7)`, entonces ahora `a == 5` y `b == 7`.

Además, al invocar a una función con argumentos posicionales, estos siempre deben tener un valor. Por ejemplo, las siguientes llamadas arrojan un error.

```
# Falta especificar el argumento b.  
sumar(7)  
# Sobra un argumento.  
sumar(7, 5, 3)
```

Ahora bien, una función puede tener argumentos con valores por defecto, de modo que, al llamarla, si no hemos especificado un valor concreto por argumento, éste toma automáticamente el que la definición le ha asignado por defecto.

```
def sumar(a, b=5):  
    return a + b
```

```
# ¡Perfecto! Imprime 12.  
print(sumar(7))  
# En este caso, b == 10.  
print(sumar(5, 10))
```

Agreguemos, para avanzar un poco más, un tercer argumento `c` con un valor por defecto.

```
def sumar(a, b=5, c=10):  
    return a + b + c
```

```
# Imprime 22 (7 + 5 + 10).  
print(sumar(7))
```

interesantes para trabajar con los argumentos. Por ejemplo, la posibilidad de crear funciones con argumentos infinitos.



Argumentos en Python



¿Qué ocurre si quiero indicar un valor para el argumento `c` mientras que `b` mantenga su valor por defecto? En ese caso, en lugar de pasar el argumento por posición, lo voy a pasar por nombre.

```
# Imprime 32 (7 + 5 + 20).
```

```
print(sumar(7, c=20))
```

La posibilidad de pasar argumentos por su nombre en la llamada a una función solo puede darse en aquellos que tengan un valor por defecto. De ahí que a estos se los conozca como **argumentos por nombre**. En estos casos, la posición de los argumentos es indistinta, de modo que el siguiente código es perfectamente válido.

```
# El orden es indistinto en los argumentos por nombre.
```

```
print(sumar(7, c=20, b=10))
```

(Nótese que, por convención, cuando especificamos el valor de un argumento por nombre no se ubican espacios alrededor del signo igual).

La única restricción es que los argumentos posicionales deben preceder a los argumentos por nombre, tanto en la definición de una función como en la llamada.

El siguiente ejemplo no es un código válido de Python.

```
# ¡¡¡Inválido!!!
```

```
def sumar(a=5, b):
```

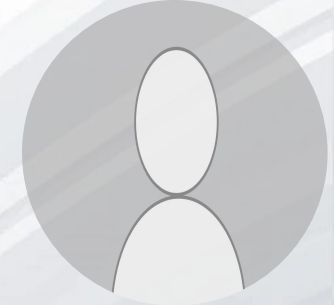
```
    return a + b
```



FUNCIONES

Python provee otras herramientas interesantes para trabajar con los argumentos. Por ejemplo, la posibilidad de crear funciones con argumentos infinitos

Paso por valor y referencia



En muchos lenguajes de programación existen los conceptos de paso por valor y por referencia que aplican a la hora de como trata una función a los parámetros que se le pasan como entrada. Su comportamiento es el siguiente:

Si usamos un parámetro pasado por **valor**, **se creará una copia local de la variable**, lo que implica que cualquier modificación sobre la misma no tendrá efecto sobre la original.

Con una variable pasada como **referencia**, **se actuará directamente sobre la variable pasada**, por lo que las modificaciones afectarán a la variable original.

*En *Python las cosas son un poco distintas, y **el comportamiento estará definido por el tipo de variable** con la que estamos tratando. Veamos un ejemplo de paso por valor.*

```
x = 10
def funcion(entrada):
    entrada = 0
funcion(x)

print(x) # 10
```



Iniciamos la x a 10 y se la pasamos a funcion(). Dentro de la función hacemos que la variable valga 0. Dado que Python trata a los int como pasados por valor, dentro de la función se crea una copia local de x, por lo que la variable original no es modificada.

No pasa lo mismo si por ejemplo x es una lista como en el siguiente ejemplo. En este caso Python lo trata como si estuviese pasada por referencia, lo que hace que se modifique la variable original. La variable original x ha sido modificada.

```
x = [10, 20, 30]
def funcion(entrada):
    entrada.append(40)

funcion(x)
print(x) # [10, 20, 30, 40]
```

Paso por valor y referencia



El ejemplo anterior nos podría llevar a pensar que si en vez de añadir un elemento a x, hacemos x=[], estaríamos destruyendo la lista original. Sin embargo esto no es cierto.

```
x = [10, 20, 30]
def funcion(entrada):
    entrada = []
```

```
funcion(x)
print(x)
# [10, 20, 30]
```

Una forma muy útil de saber lo que pasa por debajo de Python, es haciendo uso de la **función id()**. Esta función nos devuelve un identificador único para cada objeto. Volviendo al primer ejemplo podemos ver como los objetos a los que “apuntan” x y entrada son distintos.

```
x = 10
print(id(x)) # 4349704528
def funcion(entrada):
    entrada = 0
    print(id(entrada)) # 4349704208
```

```
funcion(x)
```

Sin embargo si hacemos lo mismo cuando la variable de entrada es una lista, podemos ver que en este caso el objeto con el que se trabaja dentro de la función es el mismo que tenemos fuera.

```
x = [10, 20, 30]
print(id(x)) # 4422423560
def funcion(entrada):
    entrada.append(40)
    print(id(entrada)) # 4422423560
```

```
funcion(x)
```



Argumentos “Especiales” en Python: *Args y **Kwargs en Python

Python permite definir una función con un número *variable de argumentos*, implementado gracias a los args y kwargs en Python. Vamos a suponer que queremos una función que sume un conjunto de números, pero no sabemos a priori la cantidad de números que se quieren sumar. Si por ejemplo tuviéramos tres, la función sería tan sencilla como la siguiente.

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

```
suma(2, 4, 6)  
#Salida: 12
```

El problema surge si por ejemplo queremos sumar cuatro números. Como es evidente, la siguiente llamada a la función anterior daría un error ya que estamos usando cuatro argumentos mientras que la función sólo soporta tres.

```
suma(2, 4, 6, 1)  
#TypeError: suma() takes 3 positional arguments but 4 were given
```

Introducida ya la problemática, veamos como podemos resolver este problema con ***args y **kwargs en Python**.

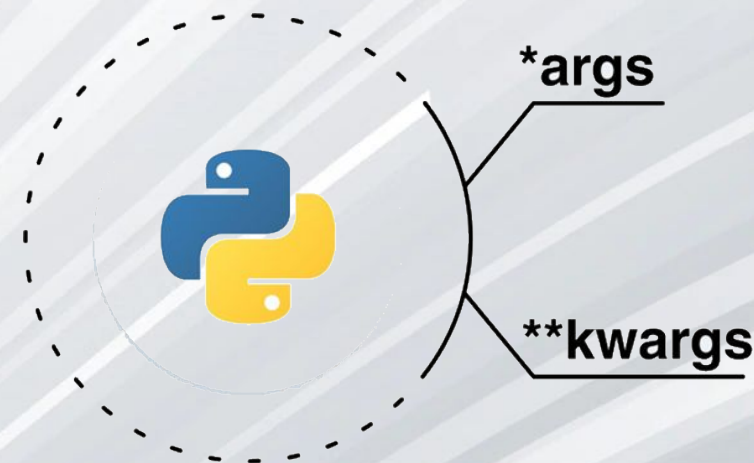
Uso de *args

Gracias a los *args en Python, podemos definir funciones cuyo número de argumentos es variable. Es decir, podemos definir funciones genéricas que no aceptan un número determinado de parámetros, sino que se “adaptan” al número de argumentos con los que son llamados.

De hecho, el args viene de arguments en Inglés, o argumentos. Haciendo uso de *args en la declaración de la función podemos hacer que el número de parámetros que acepte sea variable.

```
def suma(*args):  
    s = 0  
    for arg in args:  
        s += arg  
    return s
```

```
suma(1, 3, 4, 2)  
#Salida 10  
suma(1, 1)  
#Salida 2
```



Args y Kwargs en Python

Antes de nada, el uso del nombre **args** es totalmente arbitrario, por lo que podrías haberlo llamado como quisieras.

Es una mera convención entre los usuarios de Python y resulta frecuente darle ese nombre. Lo que si es un requisito, es usar ***** junto al nombre.

En el ejemplo anterior hemos visto como ***args** puede ser iterado, ya que en realidad es una tupla. Por lo tanto iterando la tupla podemos acceder a todos los argumentos de entrada, y en nuestro caso sumarlos y devolverlos.

Nótese que es un mero ejemplo didáctico. En realidad podríamos hacer algo como lo siguiente, lo que sería mucho más sencillo.

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

```
suma(5, 5, 3)  
#Salida 13
```

Con esto resolvemos nuestro problema inicial, en el que necesitábamos un número variable de argumentos. Sin embargo, hay otra forma que nos proporciona además un nombre asociado al argumento, con el uso de ****kwargs**. La explicamos a continuación.

Uso de **kwargs

Al igual que en ***args**, en ****kwargs** el nombre es una mera convención entre los usuarios de Python. Puedes usar cualquier otro nombre siempre y cuando respetes el ******.

En este caso, en vez de tener una tupla tenemos un **diccionario**.

Puedes verificarlo de la siguiente forma con `type()`.

```
def suma(**kwargs):  
    print(type(kwargs))
```

```
suma(x=3)  
#<class 'dict'>
```



Veamos un ejemplo.



Args y Kwargs en Python



Pero veamos un ejemplo más completo. A diferencia de `*args`, los `**kwargs` nos permiten dar un nombre a cada argumento de entrada, pudiendo acceder a ellos dentro de la función a través de un diccionario.

```
def suma(**kwargs):  
    s = 0  
    for key, value in kwargs.items():  
        print(key, "=", value)  
        s += value  
    return s
```

```
suma(a=3, b=10, c=3)  
#Salida  
#a = 3  
#b = 10  
#c = 3  
#16
```

Como podemos ver, es posible iterar los argumentos de entrada con `items()`, y podemos acceder a la clave **key** (o nombre) y el valor o **value** de cada argumento.

El uso de los `**kwargs` es muy útil si además de querer acceder al valor de las variables dentro de la función, quieres darles un nombre que de una información extra.



Args y Kwargs en Python

Mezclando *args y **kwargs

Una vez entendemos el uso de *args y **kwargs, podemos complicar las cosas un poco más. Es posible mezclar argumentos normales con *args y **kwargs dentro de la misma función. Lo único que necesitas saber es que debes definir la función en el siguiente orden:

Primero argumentos normales.
Después los *args. Y por último los **kwargs.

Veamos un ejemplo.

```
def funcion(a, b, *args, **kwargs):  
    print("a =", a)  
    print("b =", b)  
    for arg in args:  
        print("args =", arg)  
    for key, value in kwargs.items():  
        print(key, "=", value)
```

```
funcion(10, 20, 1, 2, 3, 4, x="Hola", y="Que", z="Tal")  
#Salida  
#a = 10  
#b = 20  
#args = 1  
#args = 2  
#args = 3  
#args = 4  
#x = Hola  
#y = Que  
#z = Tal
```

*args and **kwargs

Parameter 1

Parameter 2

Parameter 3

.....

***args means passing an
arbitrary number of
arguments**

****kwargs means
passing an arbitrary
number of arguments
with keywords**

Parameter 1

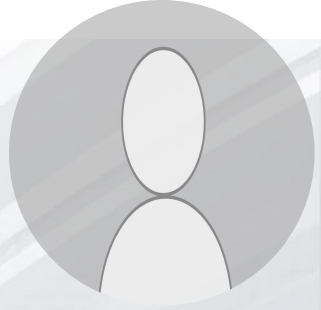
Parameter 2

Parameter 3

Parameter 4

.....

Args y Kwargs en Python



Y por último un truco que no podemos dejar sin mencionar es lo que se conoce como **tuple unpacking**. Haciendo uso de *, podemos extraer los valores de una lista o tupla, y que sean pasados como argumentos a la función.

```
def funcion(a, b, *args, **kwargs):
    print("a =", a)
    print("b =", b)
    for arg in args:
        print("args =", arg)
    for key, value in kwargs.items():
        print(key, "=", value)
```

```
args = [1, 2, 3, 4]
kwargs = {'x':"Hola", 'y':"Que", 'z':"Tal"}
```

```
funcion(10, 20, *args, **kwargs)
#Salida
#a = 10
#b = 20
#args = 1
#args = 2
#args = 3
#args = 4
#x = Hola
#y = Que
#z = Tal
```

*args and **kwargs

Parameter 1

Parameter 2

Parameter 3

.....

***args means passing an arbitrary number of arguments**

****kwargs means passing an arbitrary number of arguments with keywords**

Parameter 1

Parameter 2

Parameter 3

Parameter 4

.....

Documentación



Cuando diseñamos una función, lo ideal es documentar su comportamiento utilizando comillas triples inmediatamente luego de su definición.

```
def dup(n):  
    """  
    Retorna el doble de `n`.  
    """  
    return n * 2
```

Cuando la documentación es más bien corta, como la que hemos mostrado, podemos ubicarla por completo en la misma línea, incluso usando comillas simples.

```
def dup(n):  
    "Retorna el doble de `n`."  
    return n * 2
```

Si bien en la documentación de una función podemos escribir lo que se nos antoje, ¡no es lo mismo que un comentario! En realidad, como habrás observado, es una cadena de Python, que luego el intérprete le asignará como un atributo a la función. Para conocer el **docstring** (este es el término correcto) de una función podemos emplear la función incorporada `help()`.

```
# Imprime la documentación en pantalla.  
help(dup)
```

Esta función es especialmente útil en la consola, dado que te permite conocer el funcionamiento y la estructura de argumentos de todas las funciones incorporadas.

```
>>> help(print)  
[...]  
>>> help(int)  
[...]
```



89 Py





PYTHON

RECURSIVIDAD

RECURSIVIDAD EN PYTHON.

Recursividad

¿En qué trabajas? Estoy intentando arreglar los problemas que creé cuando intentaba arreglar los problemas que creé cuando intentaba arreglar los problemas que creé... Y así nació la recursividad.

La recursividad o recursión es un concepto que proviene de las matemáticas, y que aplicado al mundo de la programación nos permite resolver problemas o tareas donde las mismas pueden ser divididas en subtarear cuya funcionalidad es la misma. Dado que los subproblemas a resolver son de la misma naturaleza, se puede usar la misma función para resolverlos. Dicho de otra manera, una función recursiva es aquella que está definida en función de sí misma, por lo que se llama repetidamente a sí misma hasta llegar a un punto de salida.

Cualquier función recursiva tiene dos secciones de código claramente divididas:

Por un lado tenemos la sección en la que la función se llama a sí misma.

Por otro lado, tiene que existir siempre una condición en la que la función retorna sin volver a llamarse.

Es muy importante porque de lo contrario, la función se llamaría de manera indefinida.

Veamos unos ejemplos con el factorial y la serie de fibonacci.

Calcular factorial

Uno de los ejemplos mas usados para entender la recursividad, es el cálculo del factorial de un número $n!$. El factorial de un número n se define como la multiplicación de todos sus números predecesores hasta llegar a uno. Por lo tanto $5!$, leído como cinco factorial, sería $5*4*3*2*1$.

Utilizando un enfoque tradicional no recursivo, podríamos calcular el factorial de la siguiente manera.

```
def factorial_normal(n):  
    r = 1  
    i = 2  
    while i <= n:  
        r *= i  
        i += 1  
    return r
```

```
factorial_normal(5) # 120
```



Recursividad

Dado que el factorial **es una tarea que puede ser dividida en subtarear del mismo tipo** (multiplicaciones), podemos darle un enfoque recursivo y escribir la función de otra manera.

```
def factorial_recursivo(n):  
    if n == 1:  
        return 1  
    else:  
        return n * factorial_recursivo(n-1)
```

`factorial_recursivo(5) # 120`

Lo que realmente hacemos con el código anterior es llamar a la función `factorial_recursivo()` múltiples veces. Es decir, $5!$ es igual a $5 * 4!$ y $4!$ es igual a $4 * 3!$ y así sucesivamente hasta llegar a 1.

Algo muy importante a tener en cuenta es que si realizamos demasiadas llamadas a la función, podríamos llegar a tener un error del tipo `RecursionError`. Esto se debe a que todas las llamadas van apilándose y creando un contexto de ejecución, algo que podría llegar a causar un [stack overflow](#). Es por eso por lo que Python lanza ese error, para protegernos de llegar a ese punto.

Serie de Fibonacci

Otro ejemplo muy usado para ilustrar las posibilidades de la recursividad, es calcular la serie de fibonacci. Dicha serie calcula el elemento n sumando los dos anteriores $n-1 + n-2$. Se asume que los dos primeros elementos son 0 y 1.

```
def fibonacci_recursivo(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci_recursivo(n-1) + fibonacci_recursivo(n-2)
```



Podemos ver que siempre que la n sea distinta de cero o uno, se llamará recursivamente a la función, buscando los dos elementos anteriores. Cuando la n valga cero o uno se dejará de llamar a la función y se devolverá un valor concreto. Podemos calcular el elemento 7, que será 0,1,1,2,3,5,8,13, es decir, 13.

```
fibonacci_recursivo(7)  
# 13
```



PYTHON

FUNCIONES LAMBDA

FUNCIONES LAMBDA EN PYTHON.

Funciones lambda

Las funciones **lambda o anónimas** son un tipo de funciones en Python que típicamente se definen en una línea y cuyo código a ejecutar suele ser pequeño. Resulta complicado explicar las diferencias, y para que te hagas una idea de ello te dejamos con la siguiente cita sacada de la documentación oficial.

“Python lambdas are only a shorthand notation if you’re too lazy to define a function.”

Lo que significa algo así como, “las funciones lambda son simplemente una versión acortada, que puedes usar si te da pereza escribir una función”
Lo que sería una función que suma dos números como la siguiente.

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

Se podría expresar en forma de una función lambda de la siguiente manera.

```
lambda a, b : a + b
```

La primera diferencia es que **una función lambda no tiene un nombre**, y por lo tanto salvo que sea asignada a una variable, es totalmente inútil.
Para ello debemos.

```
suma = lambda a, b: a + b
```

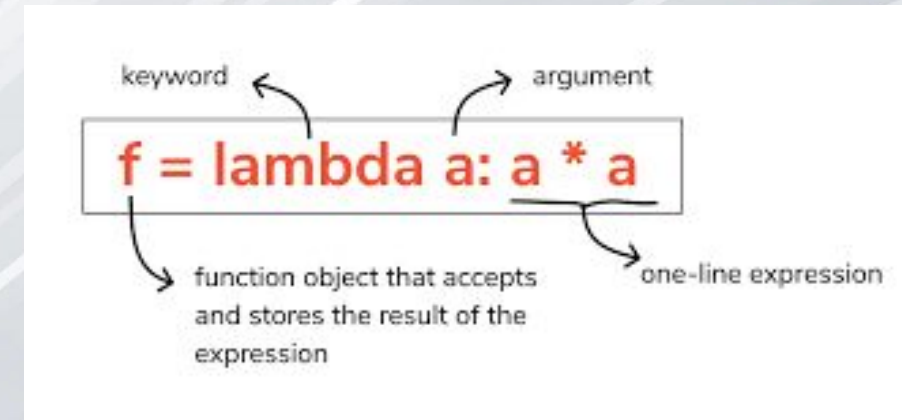
Una vez tenemos la función, es posible llamarla como si de una función normal se tratase.

```
suma(2, 4)
```

Si es una función que solo queremos usar una vez, tal vez no tenga sentido almacenarla en una variable. Es posible declarar la función y llamarla en la misma línea.

```
(lambda a, b: a + b)(2, 4)
```

Ejemplos



Funciones lambda

Una función lambda puede ser la entrada a una función normal.

```
def mi_funcion(lambda_func):  
    return lambda_func(2,4)
```

```
mi_funcion(lambda a, b: a + b)
```

Y una función normal también puede ser la entrada de una función lambda.

Nótese que son ejemplo didácticos y sin demasiada utilidad práctica per se.

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

```
(lambda a, b: mi_otra_funcion(a, b))(2, 4)
```

A pesar de que las funciones lambda tienen muchas limitaciones frente a las funciones normales, comparten gran cantidad de funcionalidades.

Es posible tener argumentos con valor asignado por defecto.

```
(lambda a, b, c=3: a + b + c)(1, 2) # 6
```

También se pueden pasar los parámetros indicando su nombre.

```
(lambda a, b, c: a + b + c)(a=1, b=2, c=3) # 6
```

Al igual que en las funciones se puede tener un número variable de argumentos haciendo uso de *, lo conocido como tuple unpacking.

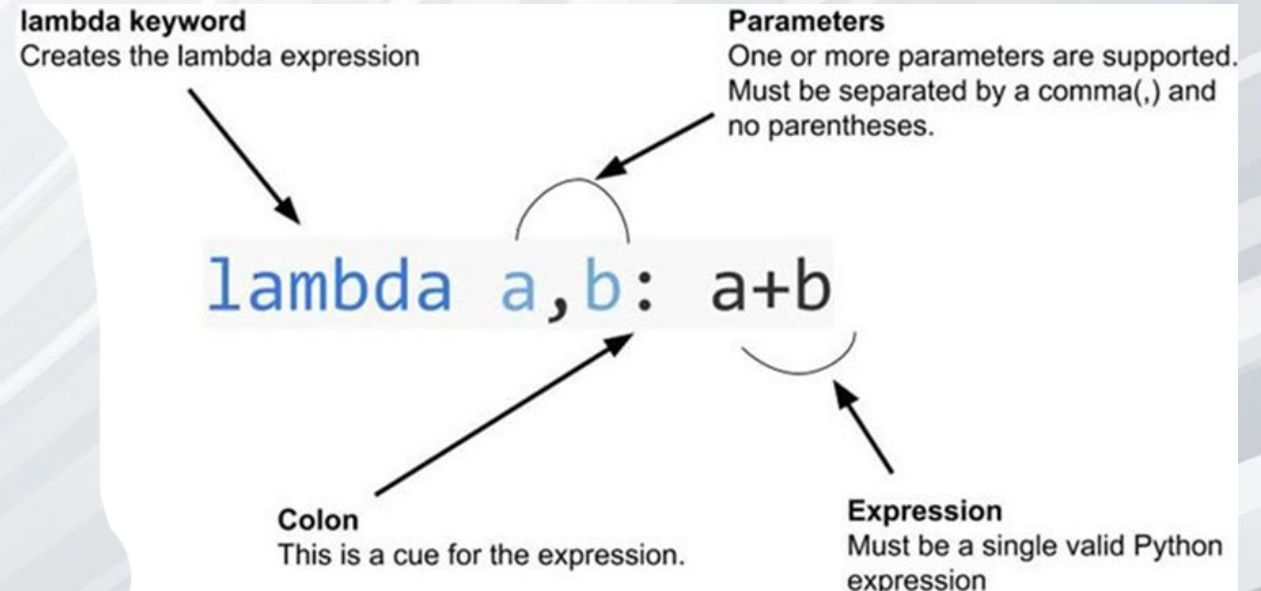
```
(lambda *args: sum(args))(1, 2, 3) # 6
```

Y si tenemos los parámetros de entrada almacenados en forma de key y value como si fuera un diccionario, también es posible llamar a la función.

```
(lambda **kwargs: sum(kwargs.values()))(a=1, b=2, c=3) # 6
```

Por último, es posible devolver más de un valor.

```
x = lambda a, b: (b, a)  
print(x(3, 9))  
# Salida (9,3)
```





PYTHON

PROGRAMACION FUNCIONAL

PROGRAMACION FUNCIONAL EN PYTHON.

Programación Funcional en Python

En pocas palabras, la programación funcional es **un paradigma de programación** distinto al tradicional estructurado u orientado a objetos al que solemos estar acostumbrados. Se basa principalmente en el uso de **funciones**, siendo las mismas prácticamente la única herramienta que el lenguaje nos ofrece.

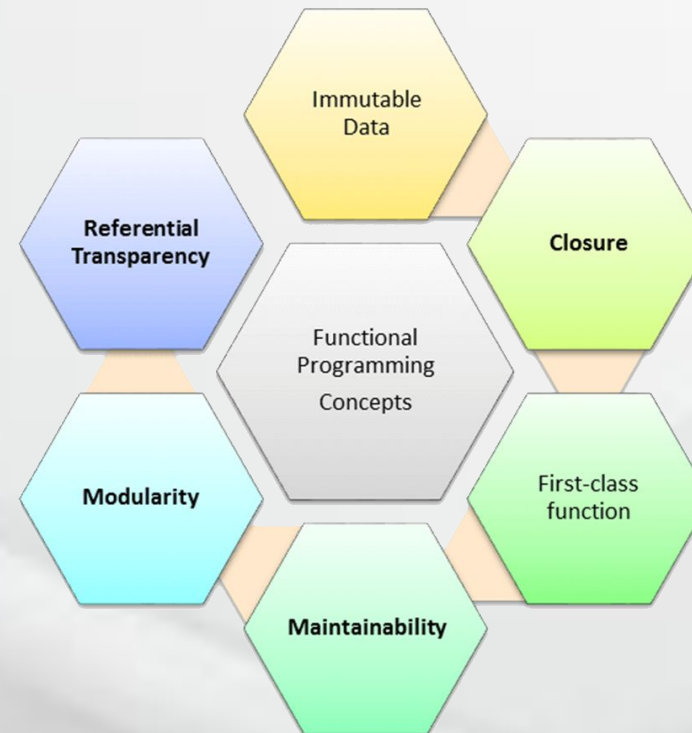
Por ello, en lenguajes puramente funcionales como Haskell no existen bucles for o while.

¿Un lenguaje de programación sin bucles? Aunque pueda parecer una locura, también tiene sus ventajas, y ofrece ciertas características muy importantes que veremos a continuación.

A pesar de que Python no es un lenguaje puramente funcional, nos ofrece algunas primitivas propias de lenguajes funcionales, como **map**, **filter** y **reduce**. Todas ellas ofrecen una alternativa al uso de bucles para resolver ciertos problemas. Veamos unos ejemplos.

Programación Funcional
en Python.

Map
filter
reduce



Programación Funcional en Python

map en Python

La función **map** toma dos entradas:

Una lista o iterable que será modificado en una nueva.

Una función, que será aplicada a cada uno de los elementos de la lista o iterable anterior.

Nos devuelve una nueva lista donde todos y cada uno de los elementos de la lista original han sido pasados por la función.

map(funcion_a_aplicar, entrada_iterable)

Imaginemos que queremos multiplicar por dos todos los elementos de una lista. La primera forma que se nos podría ocurrir sería la siguiente. Nótese que también podría usarse list comprehension, pero eso lo dejamos para otro artículo.

```
lista = [1, 2, 3, 4, 5]
lista_pordos = []
for l in lista:
    lista_pordos.append(l*2)

print(lista_pordos)
# [2, 4, 6, 8, 10]
```

Pero veamos como darle un enfoque funcional. Como hemos dicho, map toma una función y un iterable como entrada, aplicando la función a cada elemento. Entonces podemos definir una función por_dos, que pasaremos a map junto con nuestra lista inicial.

```
lista = [1, 2, 3, 4, 5]

def por_dos(x):
    return x * 2

lista_pordos = map(por_dos, lista)

print(list(lista_pordos))
# [2, 4, 6, 8, 10]
```

Como podemos observar, ya no usamos bucles. Simplemente le pasamos a map la función y la lista y ya hace el trabajo por nosotros. Dado que por_dos se trata de una función muy sencilla, es posible usar funciones lambda para compactar un poco el código, quedando lo siguiente:

```
lista = [1, 2, 3, 4, 5]
lista_pordos = map(lambda x: 2*x, lista)

print(list(lista_pordos))
# [2, 4, 6, 8, 10]
```

Programación Funcional en Python

filter en Python

La función filter también recibe una función y una lista pero el resultado es la lista inicial filtrada. Es decir, se pasa cada elemento de la lista por la función, y sólo si su resultado es True, se incluye en la nueva lista.

filter(funcion_filtrar, entrada_iterable)

Al igual que hacíamos antes, usamos las funciones lambda que nos permiten declarar y asignar una función en la misma línea de código. En el siguiente ejemplo filtramos los números pares.

```
lista = [7, 4, 16, 3, 8]
pares = filter(lambda x: x % 2 == 0, lista)
```

```
print(list(pares))
# [4, 16, 8]
```

Nótese que el siguiente código sería equivalente:

```
lista = [7, 4, 16, 3, 8]
```

```
def es_par(x):
    return x % 2 == 0
```

```
pares = filter(es_par, lista)
```

```
print(list(pares))
# [4, 16, 8]
```

Una vez más, resaltar que no estamos usando bucles. Simplemente damos la entrada y la función a aplicar a cada elemento, y filter se encarga del resto. Esta es una de las características clave de la programación funcional. Se centra más en el qué hacer que en el cómo hacerlo. Para ello se usan funciones predefinidas como las que estamos viendo, a las que sólo tenemos que pasar las entradas y hacer el trabajo por nosotros.

Programación Funcional en Python

reduce en Python

Por último, podemos usar reduce para reducir todos los elementos de la entrada a un único valor aplicando un determinado criterio. Por ejemplo, podemos sumar todos los elementos de una lista de la siguiente manera.

```
from functools import reduce
lista = [1, 2, 3, 4, 5]
suma = reduce(lambda acc, val: acc + val, lista)
print(suma) # 15
```

Lo que podría reescribirse usando la función add:

```
from operator import add
from functools import reduce
lista = [1, 2, 3, 4, 5]
suma = reduce(add, lista)
print(suma) # 15
```

O también los podemos multiplicar, modificando la función lambda.

```
from functools import reduce
lista = [1, 2, 3, 4, 5]
multiplicacion = reduce(lambda acc, val: acc * val, lista)
print(multiplicacion) # 120
```

Es importante notar que la función recibe dos argumentos: el acumulador y cada uno de los valores de la lista.

El acumulador es el valor devuelto en la iteración anterior, que va acumulando un resultado hasta que llegamos al final.

El valor es cada uno de los elementos de nuestra lista, que en nuestro caso vamos añadiendo al acumulador.

El uso de reduce es especialmente útil cuando tenemos por ejemplo una lista de diccionarios y queremos sumar todos los valores de un campo en concreto. Veamos un ejemplo donde calculamos la edad media de varias personas.

```
from functools import reduce
personas = [
    {'Nombre': 'Alicia', 'Edad': 22},
    {'Nombre': 'Bob', 'Edad': 29},
    {'Nombre': 'Charlie', 'Edad': 33}
]
suma_edad = reduce(lambda total, p: total + p['Edad'], personas, 0)
print(suma_edad/len(personas)) # 28.0
```

Programación Funcional en Python

Características de la Programación Funcional

Una vez entendido el funcionamiento de map, filter y reduce, ya tenemos unas nociones básicas de la programación funcional. Sus características más importantes son las siguientes:

Los lenguajes de programación puramente funcionales carecen de bucles for y while.

Se dice que en la programación funcional, las funciones son “ciudadanas de primera clase”, lo que nos viene a decir que prácticamente todo se hace con funciones, y no con bucles.

La programación funcional prefiere también las funciones puras, es decir, funciones sin side effects. Las funciones puras no dependen de variables externas o globales. Esto significa que para las mismas entradas, siempre se producen las mismas salidas.

Por otro lado, en la programación funcional se prefiere variables inmutables, lo que significa que una vez creadas no pueden ser modificadas. Esto es algo que verdaderamente ahorra problemas, aunque la eficiencia puede ser discutible.

En general, se considera que los lenguajes de programación funcionales son más seguros, y ofrecen formal verification.

Por último, resaltar una vez más que aunque Python no es un lenguaje puramente funcional, ofrece algunas funciones propias de lenguajes funcionales como map, filter y reduce. Si estás interesado en más, puedes echar un vistazo a el paquete itertools.

Ejemplos Programación Funcional

Dada una lista de personas almacenadas en diccionarios:

Filtrar de acuerdo al sexo

Y calcular la media

```
from functools import reduce
```

```
personas = [  
    {'Nombre': 'Alicia', 'Edad': 22, 'Sexo': 'F'},  
    {'Nombre': 'Bob', 'Edad': 25, 'Sexo': 'M'},  
    {'Nombre': 'Charlie', 'Edad': 33, 'Sexo': 'M'},  
    {'Nombre': 'Diana', 'Edad': 15, 'Sexo': 'F'},  
    {'Nombre': 'Esteban', 'Edad': 30, 'Sexo': 'M'},  
    {'Nombre': 'Federico', 'Edad': 44, 'Sexo': 'M'},  
]
```

```
hombres = list(filter(lambda x: x['Sexo'] == 'M', personas))
```

```
suma_edades = reduce(lambda suma, p: suma + p['Edad'], hombres, 0)
```

```
media_edad = suma_edades/(len(hombres))
```

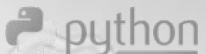
```
print(media_edad) # 33.0
```

Tal vez no muy legible, pero todo lo anterior se podrá expresar en una única línea de código.

```
media_edades = reduce(lambda suma, p: suma + p['Edad'], filter(lambda x: x['Sexo'] == 'M',  
personas), 0) / len(list(filter(lambda x: x['Sexo'] == 'M', personas))) print(media_edades) # 33.0
```



8 9 Py





PYTHON

EXCEPCIONES

EXCEPCIONES EN PYTHON.



Excepciones

Las excepciones son la herramienta que implementa Python para manejar los errores potenciales de un programa. Cuando una porción de código quiere indicar que ocurrió un error, se dice que debe **lanzar una excepción**, mientras que, cuando otra quiere saber si surgió un error y actuar en consecuencia, se dice que debe **capturarla o manejarla**. Si una excepción es lanzada y ningún código la captura, el programa finaliza.

Muchos de los operadores y funciones que hemos estado utilizando lanzan excepciones cuando ocurre algún error. Por ejemplo, habíamos visto cómo convertir una cadena a un número vía `int()`.

```
>>> int("30")
30
```

Esta función lanza una excepción cuando el argumento pasado no puede ser representado como un número entero.

```
>>> int("¡Hola, mundo!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '¡Hola, mundo!'
```

Como se observa, las excepciones están generalmente acompañadas de un mensaje que nos indica qué salió mal y en dónde (nombre del archivo y la línea). En el caso de la consola interactiva, las excepciones no manejadas se imprimen automáticamente en la pantalla, pero, como se dijo anteriormente, para programas reales un hecho tal implica la finalización del mismo.

Las excepciones son, en definitiva, **clases**. Por ello llevan la nomenclatura denominada CamelCase. `ValueError` es una excepción incorporada (definida por Python) que es lanzada cuando una función espera un valor determinado pero recibe otro.

Consideremos un ejemplo más real.

```
edad = int(input("Escribe tu edad: "))
```

```
if edad >= 18:
    print("Eres un adulto.")
else:
    print("Aún no eres un adulto.")
```

Aquí pronto vemos cuál es el problema. Si el usuario escribe cualquier cosa que no sea un número en la consola, el programa arrojará una excepción y terminará. Lo ideal sería que cuando eso ocurre, en lugar de terminar, el mismo programa vuelva a solicitar que el usuario ingrese su edad (pues bien podría haberse equivocado).

Excepciones

Lo ideal sería que cuando eso ocurre, en lugar de terminar, el mismo programa vuelva a solicitar que el usuario ingrese su edad (pues bien podría haberse equivocado). Para eso necesitamos capturar la excepción, lo cual haremos con las palabras reservadas `try` y `except`.

```
while True:
    try:
        edad = int(input("Escribe tu edad: "))
        break
    except ValueError:
        print("¡Debes ingresar un número!")

if edad >= 18:
    print("Eres un adulto.")
else:
    print("Aún no eres un adulto.")
```

Si esto te ha desconcertado un poco, analicémoslo paso a paso. Primero hemos incluido un bucle `while`, que se ejecuta infinitamente por cuanto la condición es siempre verdadera (`True`). Esto nos permitirá que el programa siga preguntando la edad hasta que el usuario ingrese un número. Para ello "envolvimos" las llamadas a `int()` e `input()` dentro de un bloque de código `try/except`. La lógica es la siguiente: cuando una excepción es capturada, la ejecución del código salta desde donde haya ocurrido (la llamada a `int()`) al bloque de código dentro de la cláusula `except`. Así, cuando `int()` lanza `ValueError`, el código salta a la llamada a `print()` sin llegar a ejecutar la palabra reservada `break`, la cual da término al bucle. Por esta razón el proceso se repite siempre que ocurra una excepción del tipo `ValueError`. Ahora bien, si la llamada a `int()` es exitosa, `break` llega a ejecutarse y el código continúa en el condicional.

El dilema también puede ser resuelto usando funciones y recursión.

```
def solicitar_edad():
    try:
        return int(input("Escribe tu edad: "))
    except ValueError:
        return solicitar_edad()

edad = solicitar_edad()
```


Excepciones

Otras excepciones incorporadas incluyen **TypeError**, **KeyError**, **IndexError**, **NameError**, **RuntimeError**, **ZeroDivisionError**.

Las irás conociendo a medida que avances con el lenguaje. Por ejemplo, **IndexError** es lanzada cuando intentamos acceder a un elemento de una lista o tupla que está por fuera de sus límites.

```
>>> lenguajes = ["Python", "C", "C++", "Java"]
```

```
>>> lenguajes[5]
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
IndexError: list index out of range
```

O bien, **KeyError**, cuando indicamos una clave que no está en un diccionario.

```
>>> d = {"Python": 1991, "C": 1972}
```

```
>>> d["Java"]
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
KeyError: 'Java'
```

Un mismo código puede lanzar múltiples excepciones. Por ejemplo, **int()**, como hemos visto, lanza **ValueError** cuando el argumento no puede ser convertido a un número entero, pero también **TypeError** cuando el argumento no es una cadena.

```
>>> int([1, 2])
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: int() argument must be a string, a bytes-like object or a number, not 'list'
```

En estos casos, podemos definir múltiples bloques de códigos para las distintas excepciones.

```
try:
```

```
    int(...) # Pseudocódigo.
```

```
except ValueError:
```

```
    print("No puede convertirse a un entero.")
```

```
except TypeError:
```

```
    print("No es una cadena.")
```



Excepciones

O bien definir el mismo bloque de código para distintas excepciones.

```
try:
    int(...)
except (ValueError, TypeError):
    print("No puede convertirse a un entero o no es una cadena.")
```

Por último, para capturar cualquier excepción:

```
try:
    int(...)
except Exception:
    print("Ocurrió un error.")
```

Lanzando una excepción

Ya sabemos cómo capturar una excepción. Pero cuando escribimos nuestras propias funciones, a menudo queremos lanzar nuestras propias excepciones. Traigamos nuevamente nuestra función sumar().

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

Queremos que únicamente sea capaz de sumar números enteros. Para ello podemos chequear el tipo de dato de los argumentos vía la función incorporada isinstance() y, en caso negativo, lanzar la excepción TypeError, que es la que más se adecúa al error que queremos expresar (esperábamos un entero pero obtuvimos otro tipo).

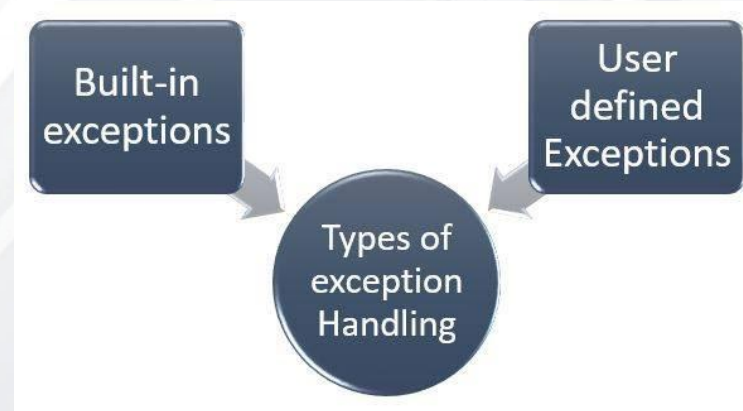
```
def sumar(a, b):
    if not isinstance(a, int) or not isinstance(b, int):
        raise TypeError("a y b tienen que ser números enteros.")
    return a + b
```

Como se observa, se emplea la palabra reservada raise seguida del nombre de la excepción y, opcionalmente, un mensaje que detalle el error ocurrido.

Definiendo una excepción

Eventualmente necesitaremos definir nuevas excepciones si ninguna de las incorporadas se adecúa a nuestros intereses. Simplemente debemos crear una clase que herede de Exception.

```
class NuevaExcepcion(Exception):
    pass
```





PYTHON

ARCHIVOS

INTRODUCCION A ARCHIVOS EN PYTHON.



Archivos

Leer y escribir archivos en Python es una tarea sumamente sencilla. No se diferencia en mucho, en efecto, de la solución de otros lenguajes: todos proveen una API bastante similar.

Para abrir un archivo emplearemos la función incorporada `open()`. Luego, podremos acceder a su contenido vía el método `read()` y finalmente cerrar el fichero vía `close()`.

```
f = open("archivo.txt")
print(f.read())
f.close()
```

El primer argumento es una cadena que indica la ruta del archivo que queremos abrir, la cual puede ser absoluta (p. ej. `C:\Documentos\archivo.txt`) o bien relativa, como se observa en el ejemplo.

Para escribir en un archivo, utilizamos el método `write()`. Pero primero debemos indicarle a `open()` que queremos abrir el archivo como escritura, con un segundo argumento.

```
f = open("archivo.txt", "w")
f.write("¡Hola, mundo!")
f.close()
```

Nótese que este código, al escribir, reemplaza todo contenido anterior. Para preservarlo y, en su lugar, escribir al final del archivo, debemos indicar un tercer modo de apertura.

```
f = open("archivo.txt", "a")
# Esto se agrega al final del archivo.
f.write("¡Hola, mundo!")
f.close()
```

Si bien no hemos hablado de la palabra reservada `with`, considero un deber comentarte esta otra forma de trabajar con archivos.

```
with open("archivo.txt") as f:
    print(f.read())
```

Este ejemplo es similar al primero, con la diferencia que Python se encargará de llamar a `close()` cuando hayas terminado de usar el archivo, ¡incluso si ocurre una excepción! Por ello se considera la opción más recomendada.

Otros métodos de los ficheros incluyen `seek()`, `readline()` y `readlines()`

Archivos (Adicional)

La entrada y salida de ficheros (comúnmente llamada File I/O) es una herramienta que Python, como la mayoría de los lenguajes, nos brinda de forma estándar. Su utilización es similar a la del lenguaje C, aunque añade mayores características. Puedes encontrar la documentación oficial del tema Reading and Writing Files en este enlace.

El primer paso para introducirnos es saber cómo abrir un determinado archivo. Para esto se utiliza la función `open`, que toma como primer argumento el nombre de un fichero.

```
open("archivo.txt")
```

No está de más aclarar que esto no significa ejecutar el programa por defecto para editar el fichero, sino cargarlo en la memoria para poder realizar operaciones de lectura y escritura.

En caso de no especificarse la ruta completa, se utilizará la actual (donde se encuentre el script).

Como segundo argumento, `open` espera el modo (como una cadena) en el que se abrirá el fichero. Esto indica si se utilizará para lectura, escritura, ambas, entre otras.

```
open("archivo.txt", "r")
```

La `r` indica el modo lectura. Si se intentara utilizar la función `write` para escribir algo, se lanzaría la excepción `IOError`. A continuación los distintos modos.

- `r` – Lectura únicamente.
- `w` – Escritura únicamente, reemplazando el contenido actual del archivo o bien creándolo si es inexistente.
- `a` – Escritura únicamente, manteniendo el contenido actual y añadiendo los datos al final del archivo.
- `w+`, `r+` o `a+` – Lectura y escritura.

El signo `+` permite realizar ambas operaciones. La diferencia entre `w+` y `r+` consiste en que la primera opción borra el contenido anterior antes de escribir nuevos datos, y crea el archivo en caso de ser inexistente. `a+` se comporta de manera similar, aunque añade los datos al final del archivo.

Todas las opciones anteriores pueden combinarse con una `'b'` (de binary), que consiste en leer o escribir datos binarios. Esta opción es válida únicamente en sistemas Microsoft Windows, que hacen una distinción entre archivos de texto y binarios. En el resto de las plataformas, es simplemente ignorada. Ejemplos: `rb`, `wb`, `ab+`, `rb+`, `wb+`.

Archivos (Adicional)

Por último, `open()` retorna una instancia de la clase `file`, desde la cual podremos ejecutar diversas operaciones.

```
f = open("archivo.txt", "w")
```

En este ejemplo he abierto el fichero `archivo.txt` que, en caso de no existir, será automáticamente creado en el directorio actual. A continuación escribiré algunos datos utilizando la función `file.write`.

```
f.write("Esto es un texto.")
```

Reiteradas llamadas a esta función añaden texto al archivo hasta que sea cerrado.

```
f.write("Esto es una linea.\n")
```

```
f.write("Esto es otra linea.\n")
```

```
f.write("Esta es la tercer linea.\n")
```

Nótese el uso del carácter `'\n'` para indicar el salto de línea.

Por último, cerramos el archivo.

```
f.close()
```

Con respecto a la lectura, las tres funciones disponibles son `read`, `readline` y `readlines`.

```
f = open("archivo.txt", "r")
```

```
content = f.read()
```

```
print(content)
```

```
f.close()
```

Imprime:

```
Esto es una linea.
```

```
Esto es otra linea.
```

```
Esta es la tercer linea.
```

Archivos (Adicional)

Una segunda llamada a `f.read()` retornaría una cadena vacía, debido a que ya se ha obtenido todo el contenido.

```
print(f.readline())  
print(f.readline())  
Imprime:
```

Esto es una linea.

Esto es otra linea.

Nótse el carácter de nueva línea al final de la cadena retornada.

Para iterar entre todas las líneas de un archivo, ambos ejemplos resultan similares:

```
for line in f.readlines():  
    print(line)  
for line in f:  
    print(line)  
Resultado:
```

Esto es una linea.

Esto es otra linea.

Esta es la tercer linea.

Archivos (Adicional)

`file.seek()` permite posicionarse en un determinado lugar del fichero. Ejemplo:

```
f.seek(30)
print(f.read())
```

Comienza a leer desde la posición 30, por lo que imprime:

ra línea.

Esta es la tercer línea.

A partir de Python 2.5 puede utilizarse `with` para asegurar que el archivo sea cerrado al finalizar su utilización, incluso si se produce un error.

`with open("archivo.txt", "r") as f:`

```
    content = f.read()
print(f.closed) # True
```

Es equivalente a:

```
try:
```

```
    f = open("archivo.txt", "r")
    content = f.read()
```

```
finally:
```

```
    f.close()
```





PYTHON

MODULOS Y PAQUETES

INTRODUCCION A PAQUETES EN PYTHON.



Módulos y paquetes

Un módulo es una unidad de software que provee una función. Podemos exportar librerías como módulos para integrarlas a nuestro código, o crear módulos para dividir nuestro código en partes y que sea más fácil de manejar. Un paquete es un archivo o folder que puede contener uno o más módulos. Por ejemplo, podemos definir un módulo `mimodulo.py` con dos funciones `suma()` y `resta()`.

```
# mimodulo.py
def suma(a, b):
    return a + b
```

```
def resta(a, b):
    return a - b
```

Una vez definido, dicho módulo puede ser usado o importado en otro fichero, como mostramos a continuación. Usando `import` podemos importar todo el contenido.

```
# otromodulo.py
import mimodulo
```

```
print(mimodulo.suma(4, 3)) # 7
print(mimodulo.resta(10, 9)) # 1
```

También podemos importar únicamente los componentes que nos interesen como mostramos a continuación.

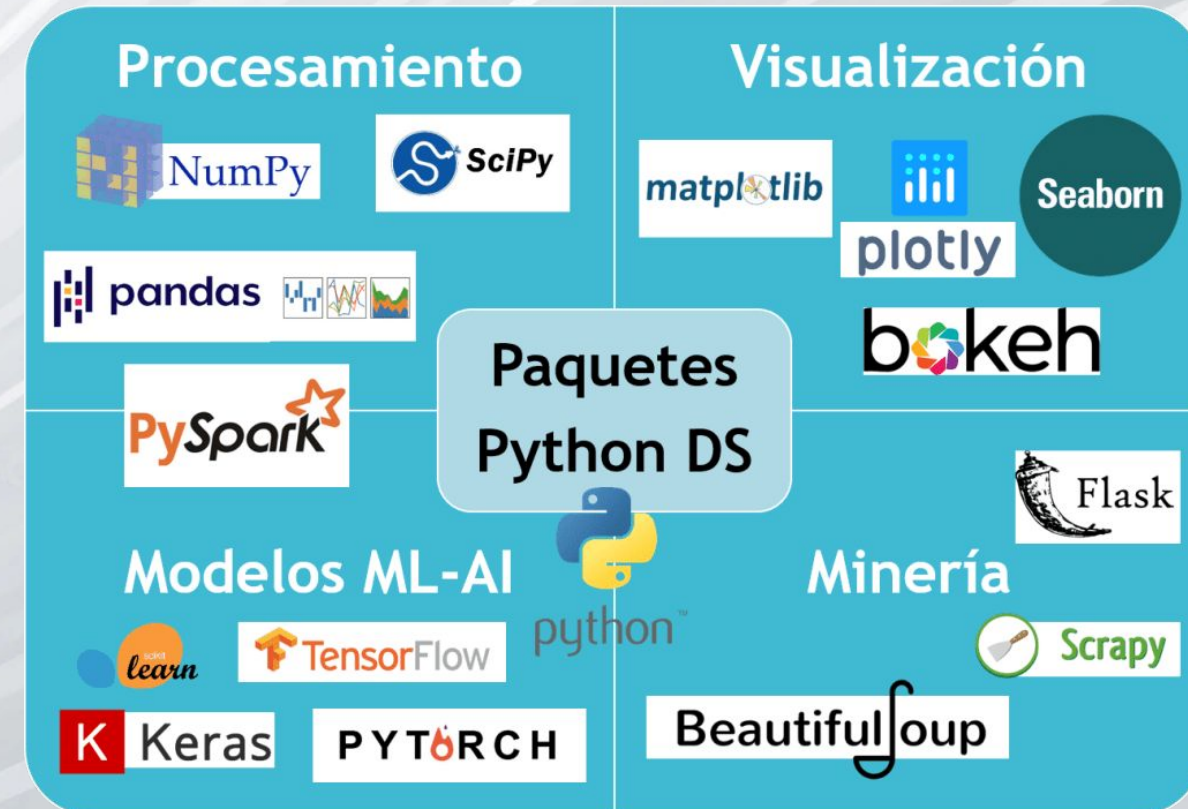
```
from mimodulo import suma, resta
```

```
print(suma(4, 3)) # 7
print(resta(10, 9)) # 1
```

Por último, podemos importar todo el módulo haciendo uso de `*`, sin necesidad de usar `mimodulo.*`.

```
from mimodulo import *
```

```
print(suma(4, 3)) # 7
print(resta(10, 9)) # 1
```



Módulos y paquetes

Rutas y Uso de sys.path

Normalmente los módulos que importamos están en la misma carpeta, pero es posible acceder también a módulos ubicados en una subcarpeta. Imaginemos la siguiente estructura:

```
.
├── ejemplo.py
├── carpeta
│   └── modulo.py
```

Donde modulo.py contiene lo siguiente:

```
# modulo.py
def hola():
    print("Hola")
```

Desde nuestro ejemplo.py, podemos importar el módulo modulo.py de la siguiente manera:

```
from carpeta.modulo import *
print(hola())
# Hola
```

Es importante notar que Python busca los módulos en las rutas indicadas por el sys.path. Es decir, cuando se importa un módulo, lo intenta buscar en dichas carpetas. Puedes ver tu sys.path de la siguiente manera:

```
import sys
print(sys.path)
```

Como es obvio, verás que la carpeta de tu proyecto está incluida, pero ¿y si queremos importar un módulo en una ubicación distinta? Pues bien, podemos añadir al sys.path la ruta en la que queremos que Python busque.

```
import sys
sys.path.append(r'ruta/de/tu/modulo')
```

Una vez realizado esto, los módulos contenidos en dicha carpeta podrán ser importados sin problema como hemos visto anteriormente.

Módulos y paquetes

Cambiando los Nombres con as

Por otro lado, es posible cambiar el nombre del módulo usando as. Imaginemos que tenemos un módulo moduloconnombrrelargo.py.

```
# moduloconnombrrelargo.py
hola = "hola"
```

En vez de usar el siguiente import, tal vez queramos asignar un nombre más corto al módulo.

```
import moduloconnombrrelargo
print(moduloconnombrrelargo.hola)
```

Podemos hacerlo de la siguiente manera con as:

```
import moduloconnombrrelargo as m
print(m.hola)
## Listando dir
```

La función dir() nos permite ver los nombres (variables, funciones, clases, etc) existentes en nuestro namespace. Si probamos en un módulo vacío, podemos ver como tenemos varios nombres rodeados de __. Se trata de nombres que Python crea por debajo.

```
print(dir())
# ['__annotations__', '__builtins__', '__cached__', '__doc__',
#  '__file__', '__loader__', '__name__', '__package__', '__spec__']
```

Por ejemplo, __file__ es creado automáticamente y alberga el nombre del fichero .py.

```
print(__file__)
#/tu/ruta/tufichero.py
```


Módulos y paquetes

Cambiando los Nombres con as

Imaginemos ahora que tenemos alguna variable y función definida en nuestro script. Como era de esperar, `dir()` ahora nos muestra también los nuevos nombres que hemos creado, y que por supuesto pueden ser usados.

```
mi_variable = "Python"
def mi_funcion():
    pass
print(dir())
```

```
['__annotations__', '__builtins__', '__cached__', '__doc__',
 '__file__', '__loader__', '__name__', '__package__', '__spec__',
 'mi_funcion', 'mi_variable']
```

Por último, vamos a importar el contenido de un módulo externo. Podemos ver que en el namespace tenemos también los nombres `resta` y `suma`, que han sido tomados de `mimodulo`.

```
from mimodulo import *
print(dir())
```

```
['__annotations__', '__builtins__', '__cached__',
 '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'resta', 'suma']
```

El uso de `dir()` también acepta parámetros de entrada, por lo que podemos por ejemplo pasar nuestro modulo y nos dará más información sobre lo que contiene.

```
import mimodulo
```

```
print(dir(mimodulo))
['__builtins__', '__cached__', '__doc__',
 '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'resta', 'suma']
print(mimodulo.__name__)
# mimodulo
```

```
print(mimodulo.__file__)
# /tu/ruta/mimodulo.py
```


Módulos y paquetes

Excepción ImportError

Importar un módulo puede lanzar una excepción, cuando se intenta importar un módulo que no ha sido encontrado. Se trata de `ModuleNotFoundError`.

```
import moduloquenoexiste  
# ModuleNotFoundError: No module named 'moduloquenoexiste'  
Dicha excepción puede ser capturada para evitar la interrupción del programa.
```

```
try:  
    import moduloquenoexiste  
except ModuleNotFoundError as e:  
    print("Hubo un error:", e)
```

Módulos y paquetes

Módulos y Función Main

Un problema muy recurrente es cuando creamos un módulo con una función como en el siguiente ejemplo, y añadimos algunas sentencias a ejecutar.

```
# modulo.py
```

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

```
c = suma(1, 2)  
print("La suma es:", c)
```

Si en otro módulo importamos nuestro modulo.py, tal como está nuestro código el contenido se ejecutará, y esto puede no ser lo que queramos.

```
# otromodulo.py  
import modulo
```

```
# Salida: La suma es: 3
```

Dependiendo de la situación, puede ser importante especificar que únicamente queremos que se ejecute el código si el módulo es el `__main__`. Con la siguiente modificación, si hacemos `import modulo` desde otro módulo, este fragmento ya no se ejecutará al ser el módulo main otro.

```
# modulo.py
```

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

```
if (__name__ == '__main__'):  
    c = suma(1, 2)  
    print("La suma es:", c)
```

Módulos y paquetes

Recargando Módulos

Es importante notar que los módulos solamente son cargados una vez. Es decir, no importa el número de veces que llamemos a `import mimodulo`, que sólo se importará una vez. Imaginemos que tenemos el siguiente módulo que imprime el siguiente contenido cuando es importado.

```
# mimodulo.py
```

```
print("Importando mimodulo")
```

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

```
def resta(a, b):  
    return a - b
```

A pesar de que llamamos tres veces al `import`, sólo vemos una única vez el contenido del `print`.

```
import mimodulo  
import mimodulo  
import mimodulo  
# Importando mimodulo
```

Si queremos que el módulo sea recargado, tenemos que ser explícitos, haciendo uso de `reload`.

```
import mimodulo  
import importlib  
importlib.reload(mimodulo)  
importlib.reload(mimodulo)
```





PYTHON

TESTING & FRAMEWORK TEST

INTRODUCCION AL TESTING EN PYTHON.

TESTING PYTHON

```
test_unittest.py x
1 import unittest
2 import inc_dec
3
4 class TestIncrementDecrement(unittest.TestCase):
5     def test_increment(self):
6         self.assertEqual(inc_dec.increment(3), 4)
7
8     def test_decrement(self):
9         self.assertEqual(inc_dec.decrement(3), 4)
10
11 if __name__ == '__main__':
12     unittest.main()
13
```

Dentro de la ingeniería software y la programación en general, el testing es una de las partes más importantes que nos encontraremos en casi cualquier proyecto. De hecho es común dedicar más tiempo a probar que el código funciona correctamente que a escribirlo. A continuación veremos las formas más comunes de testear el código en Python, desde lo más básico a conceptos algo más avanzados.

Testing Python

Tests Manuales y Tests Automatizados

Dado que hemos ejecutado tests sobre tu código sin darte cuenta. De acuerdo a su forma de ejecución, los podemos clasificar en:

- **Tests manuales**: Son tests ejecutados manualmente por una persona, probando diferentes combinaciones y viendo que el comportamiento del código es el esperado. Sin duda los has realizado alguna vez.
- **Tests automáticos**: Se trata de código que testea que otro código se comporta correctamente. La ejecución es automática, y permite ejecutar gran cantidad de verificaciones en muy poco tiempo. Es la forma más común, pero no siempre es posible automatizar todo.

Imaginemos que hemos escrito una función que calcula la media de los valores que se pasan en una lista como entrada.

```
def calcula_media(*args):  
    return(sum(*args)/len(*args))
```

A nadie se le ocurriría publicar nuestra función `calcula_media` sin haber hecho alguna verificación anteriormente. Podemos por ejemplo probar con los siguientes datos y ver si la función hace lo que se espera de ella. Al hacer esto ya estaríamos probando manualmente nuestro código.

```
print(calcula_media([3, 7, 5]))  
# 5.0
```

```
print(calcula_media([30, 0]))  
# 15.0
```

Con bases de código pequeñas y donde sólo trabajemos nosotros, tal vez sea suficiente, pero a medida que el proyecto crece puede no ser suficiente. ¿Qué pasa si alguien modifica nuestra función y se olvida de testear que funciona correctamente? Nuestra función habría dejado de funcionar y nadie se habría enterado.

Es aquí donde los **test automáticos** nos pueden ayudar. Python nos ofrece herramientas que nos permiten escribir tests que son ejecutados automáticamente, y que si fallan darán un error, alertando al programador de que ha “roto” algo. Podemos hacer esto con `assert`, donde identificamos dos partes claramente:

Testing Python

Tests Manuales y Tests Automatizados

Es aquí donde los test automáticos nos pueden ayudar. Python nos ofrece herramientas que nos permiten escribir tests que son ejecutados automáticamente, y que si fallan darán un error, alertando al programador de que ha “roto” algo. Podemos hacer esto con `assert`, donde identificamos dos partes claramente:

Por un lado tenemos la llamada a la función que queremos testear, que devuelve un resultado.

Por otro lado tenemos el resultado esperado, que comparamos con el resultado devuelto por la función. Si no es igual, se lanza un error.

```
assert(calcula_medio([3, 7, 5]) == 5.0)
assert(calcula_medio([30, 0]) == 15.0)
```

Nótese que los valores de 5 y 15 los hemos calculado manualmente, y corresponden con la media de 3,7,5 y 30,0 respectivamente.

Si por cualquier motivo alguien rompe nuestra función `calcula_medio()`, cuando los tests se ejecuten lanzaran una excepción.

Traceback (most recent call last):

File "ejemplo.py", line 7, in <module>

assert((calcula_medio([30, 0]) == 15.0))

AssertionError

En proyectos grandes, es común que antes de permitirnos hacer merge de nuestro código en master, se nos obligue a ejecutar un conjunto de tests automatizados. Si todos pasan, se asumirá que nuestro código funciona y que no hemos “roto” nada, por lo que tendremos el visto bueno.

Visto esto, tal vez pueda parecer que los test automatizados son lo mejor, sin embargo no siempre se pueden automatizar los tests. Si por ejemplo estamos trabajando con interfaces de usuario, es posible que no podamos automatizarlos, ya que se sigue necesitando a un humano que verifique los cambios visualmente.



Testing Python

Tests Unitarios en Python con unittest

Aunque el uso de `assert()` puede ser suficiente para nuestros tests, a veces se nos queda corto y necesitamos librerías como `unittest`, que ofrecen alguna que otra funcionalidad que nos hará la vida más fácil. Veamos un ejemplo. Recordemos nuestra función `calcula_media`, que es la que queremos testear.

```
# funciones.py
def calcula_media(*args):
    return(sum(*args)/len(*args))
```

Podemos usar **unittest** para crear varios tests que verifiquen que nuestra función funciona correctamente. Aunque la estructura de un conjunto de tests se puede complicar más, la estructura será siempre muy similar a la siguiente:

Creamos una clase `Test<NombreDeLoQueSePrueba>` que hereda de `unittest.TestCase`.

Definimos varios tests como métodos de la clase, usando `test_<NombreDelTest>` para nombrarlos.

En cada test ejecutamos las comprobaciones necesarias, usando `assertEqual` en vez de `assert`, pero su comportamiento es totalmente análogo.

```
# tests.py
from funciones import calcula_media
import unittest

class TestCalculaMedia(unittest.TestCase):
    def test_1(self):
        resultado = calcula_media([10, 10, 10])
        self.assertEqual(resultado, 10)

    def test_2(self):
        resultado = calcula_media([5, 3, 4])
        self.assertEqual(resultado, 4)

if __name__ == '__main__':
    unittest.main()
```



**Unit Testing With
Python Unittest**

Testing Python

Tests Unitarios en Python con unittest

Si ejecutamos el código anterior, obtendremos el siguiente resultado. Esta es una de las ventajas de unittest, ya que nos muestra información sobre los tests ejecutados, el tiempo que ha tardado y los resultados.

```
Ran 2 tests in 0.006s
```

OK

Por otro lado, usando -v podemos obtener más información sobre cada test ejecutado con su resultado individualmente. Si tenemos gran cantidad de tests suele ser recomendable usarla, ya que será más fácil localizar los tests que han fallado.

```
$ python -m unittest -v tests
```

```
test_1 (tests.TestCalculaMedia) ... ok
test_2 (tests.TestCalculaMedia) ... ok
```

```
-----
Ran 2 tests in 0.000s
```

OK

Por último, si tenemos varios ficheros de test, podemos usar discover, para decirle a Python que busque en la carpeta todos los tests y los ejecute.

```
$ python -m unittest discover -v
```

```
# tests.py
from funciones import calcula_media
import unittest

class TestCalculaMedia(unittest.TestCase):
    def test_1(self):
        resultado = calcula_media([10, 10, 10])
        self.assertEqual(resultado, 10)

    def test_2(self):
        resultado = calcula_media([5, 3, 4])
        self.assertEqual(resultado, 4)

if __name__ == '__main__':
    unittest.main()
```

Testing Python

Otras comprobaciones en unittest

Anteriormente hemos visto el uso de `.assertEqual(a, b)` que simplemente verifica que dos valores `a` y `b` son iguales, y de lo contrario da error. Sin embargo `unittest` nos ofrece un amplio abanico de opciones. Nótese que existen algunas variaciones usando “not”, como `assertNotIn()`:

- `.assertEqual(a, b)`: Verifica la igualdad de ambos valores.
- `.assertTrue(x)`: Verifica que el valor es `True`.
- `.assertFalse(x)`: Verifica que el valor es `False`.
- `.assertIs(a, b)`: Verifica que ambas variables son la misma (ver operador `is`).
- `.assertIsNone(x)`: Verifica que el valor es `None`.
- `.assertIn(a, b)`: Verifica que `a` pertenece al iterable `b` (ver operador `in`).
- `.assertIsInstance(a, b)`: Verifica que `a` es una instancia de `b`.
- `.assertRaises(x)`: Verifica que se lanza una excepción.

```
import unittest
class TestEjemplos(unittest.TestCase):
    def test_in(self):
        # Ok ya que 1 esta contenido en [1, 2, 3]
        self.assertIn(1, [1, 2, 3])

    def test_is(self):
        a = [1, 2, 3]
        b = a
        # Ok ya que son el mismo objeto
        self.assertIs(a, b)

    def test_excepcion(self):
        # Dividir 0/0 da error, pero es lo esperado por el test
        with self.assertRaises(ZeroDivisionError):
            x = 0/0
```



```
$ python -m unittest -v tests
```

```
test_excepcion (tests.TestEjemplos) ... ok
test_in (tests.TestEjemplos) ... ok
test_is (tests.TestEjemplos) ... ok
```

```
Ran 3 tests in 0.000s
```

```
OK
```

Testing Python

Usando setUp y tearDown

Otra de las ventajas de usar unittest, es que nos ofrece la posibilidad de definir funciones comunes que son ejecutadas antes y después de cada test. Estos métodos son setUp() y tearDown().

```
import unittest
class TestEjemplos(unittest.TestCase):
    def setUp(self):
        print("Entra setUp")
    def tearDown(self):
        print("Entra tearDown")

    def test_1(self):
        print("Test: test_1")
    def test_2(self):
        print("Test: test_2")
```

Siendo el resultado el siguiente. Podemos ver que hace una especie de sandwich con cada test, metiéndolo entre setUp y tearDown. Nótese que si ambas funciones realizan siempre lo mismo, tal vez se pueda usar un TestSuite con una función común para todos los tests.

```
$ python -m unittest -v tests
```

```
test_1 (tests.TestEjemplos) ... Entra setUp
Test: test_1
Entra tearDown
ok
test_2 (tests.TestEjemplos) ... Entra setUp
Test: test_2
Entra tearDown
ok
```

```
Ran 2 tests in 0.000s
OK
```


Testing Python

Usando setUp y tearDown

Otra de las ventajas de usar unittest, es que nos ofrece la posibilidad de definir funciones comunes que son ejecutadas antes y después de cada test. Estos métodos son setUp() y tearDown().

```
import unittest
class TestEjemplos(unittest.TestCase):
    def setUp(self):
        print("Entra setUp")
    def tearDown(self):
        print("Entra tearDown")

    def test_1(self):
        print("Test: test_1")
    def test_2(self):
        print("Test: test_2")
```

Siendo el resultado el siguiente. Podemos ver que hace una especie de sandwich con cada test, metiéndolo entre setUp y tearDown. Nótese que si ambas funciones realizan siempre lo mismo, tal vez se pueda usar un TestSuite con una función común para todos los tests.

```
$ python -m unittest -v tests
```


```
test_1 (tests.TestEjemplos) ... Entra setUp
Test: test_1
Entra tearDown
ok
test_2 (tests.TestEjemplos) ... Entra setUp
Test: test_2
Entra tearDown
ok
```

```
Ran 2 tests in 0.000s
OK
```



26 Testing Py





Language - Agnostic

Language-agnostic programing

La programación o secuencias de comandos independientes del lenguaje es un paradigma de desarrollo de software en el que se elige un lenguaje en particular debido a su idoneidad para una tarea en particular, y no simplemente por el conjunto de habilidades disponibles dentro de un equipo de desarrollo.

Modern Portfolio Designed



THANK YOU

Let's Python Code...