FUNCIONES EN PYTHON

Una *función* es una secuencia de sentencias que realizan una operación y que reciben un nombre.
Cuando se define una función, se especifica el nombre y la secuencia de sentencias**.**
Una función tiene las siguientes características:

1. La palabra clave ***def***
2. Un nombre de función
3. Paréntesis'**()**', y dentro de los paréntesis los parámetros de entrada, aunque los parámetros de entrada sean opcionales.
4. Dos puntos'**:**'
5. Algún bloque de código para ejecutar
6. Una sentencia de retorno, **return** (opcional)

Una función es un bloque de código que solo se ejecuta cuando se llama.
Son útiles para reutilizar código de manera eficiente.

Imagine que realiza una tarea muchas veces de forma sistemática una y otra vez.
Lo que haría seria copiar el mismo trozo de código una y otra vez. Pero si tuviera algún error o quisiera añadir alguna mejora, debería cambiar todas las partes donde lo ha escrito, haciendo que sea pesado y tedioso.

Con las funciones evitamos este problema, ya que sólo debemos tocar una parte del código, no el resto del código.

Sintaxis:

```
def nombreFuncion([parámetro/s]):
        cuerpo
……
[return (valor)]
```

#Se deben respetar las tabulaciones, de lo contrario daría error

- Grupo de código que realiza una tarea específica
- Puede o no devolver valores
- Puede o no tener parámetros
- Función y método son "Sinónimos"
- Parámetros separados por comas
- El cuerpo va indentado
- Return es opcional

.

Invocación: Para llamar a una función, use el nombre de la función seguido de paréntesis

```
nombreFuncion() nombreFuncion(parametro/s)
```

La información se puede pasar a funciones como argumentos.
Los argumentos se especifican después del nombre de la función, entre paréntesis. Se puede agregar tantos argumentos como se desee, solo se debe separar con una coma.

Utilidad principal: reutilización y modularización del código

Podemos crear dentro de la función nuestras propias variables.
También, devolver datos que podemos usar en nuestro programa principal. Para ello se deberá usar la palabra reservada **return**. Si no desea devolver ningún valor, se devolverá el valor predeterminado **None**.

Para llamar a una función, simplemente debemos poner el nombre de la función y pasarles los valores de los parámetros (pueden ser variables o valores que nosotros le demos directamente). Si devuelve algo debemos mostrar o guardar ese valor devuelto

Los términos ***parámetro* y *argumento*** se pueden usar para lo mismo: **información que se pasa a una función.**
Un **parámetro** es la variable listada entre paréntesis en la definición de la función.
Un **argumento** es el valor que se envía a la función cuando se llama.

Al definir una función los valores los cuales se reciben se denominan parámetros, pero durante la llamada los valores que se envían se denominan argumentos.

```
def función(nombre):
    print(nombre + "Martínez")
función('Marina')
función('Pedro')
```

*Devolverá**: Marina Martínez,***
Pedro Martínez

****

```
def resta(a, b):
… return a - b
…
resta(30, 10)
```

*Devolverá**:20***

Si una función espera 2 argumentos, debe llamar a la función con 2 argumentos, ni más ni menos, de lo contrario devolverá error.

Funciones incorporadas del lenguaje

Ej. print()
Input()
Float()

Funciones definidas por el usuario

Ej miFuncion():
Potencia(x,y)
Operación(p,q)

Los parámetros son siempre **"POR REFERENCIA"**

Argumentos arbitrarios, * argumentos

Si desconoce cuántos argumentos se pasarán a su función, agregue un *****antes del nombre del parámetro en la definición de la función.
La función recibirá una tupla de argumentos y podrá acceder a los elementos.

Argumentos de palabras clave

Se pueden enviar argumentos con la sintaxis ***clave = valor***. De esta forma no importa el orden de los argumentos.

```
def mi_función2(hijo1, hijo2, hijo3):
        print(“El mayor de los hijos es: “ + hijo2)

mi_función((hijo1 = "Marina", hijo2 = "Pedro", hijo3 = "Luis")
```

Devolverá: **El mayor de los hijos es Pedro**

Pasar una lista como argumento

Puede enviar cualquier tipo de datos de argumento a una función (cadena, número, lista, diccionario, etc.), y se tratará como el mismo tipo de datos dentro de la función.
Por ejemplo, si envía una Lista como argumento, seguirá siendo una Lista cuando llegue a la función:

```
def mi_función(comida):
    for x in comida:
        print(x)
    frutas = ["manzana", "banana", "cereza"]
mi_función(frutas)
```

Devolverá**: manzana**
banana
cereza

Valores devueltos
Para permitir que una función devuelva un valor, use la **return** declaración:

```
def my_function(x):
  return 5 * x

print(miFunción (3))
print(miFunción (5))
print(miFunción(9))
```

Devolverá:***15***
25
45

Funciones yield (generadores)

yield es una orden muy similar a un return, con una gran diferencia, **yield** pausará la ejecución de tu **función** y guardará el estado de la misma hasta que decidas usarla de nuevo

TRABAJAMOS EN UN PROYECTO

Es importante considerar la Opción de trabajar con “Librerías” para facilitar el trabajo
Para eso crearemos una carpeta con el nombre del proyecto a donde guardaremos los archivos que habremos de usar.
En uno de esos archivos declararemos todas las funciones que formarán parte del mismo y desde allí se podrán importar todas o alguna/as funciones que se necesiten, es más cómodo y no hay necesidad de copiar y pegar.

carpeta: ***PROYECTO***

archivos de la carpeta: { *Proyecto1*, *Proyecto_Funciones*

CONTROL DE ERRORES

SENTENCIAS try y except DE PYTHON: CÓMO MANEJAR EXCEPCIONES EN PYTHON

Son errores que ocurren durante la ejecución, es decir que la sintaxis y la semántica está bien, pero algo ajeno a la aplicación produce un error.

```
try:
        # Codigo a ejecutar
        # Pero podria haber errores en este bloque

except <tipo de error>:
        # Haz esto para manejar la excepcion
        # El bloque except se ejecutara si el bloque try lanza un error

else:
        # Esto se ejecutara si el bloque try se ejecuta sin errores

finally:
        # Este bloque se ejecutara siempre
```

USO DE CADA UNO DE ESTOS BLOQUES:

- EL bloque **try** es el bloque con las sentencias que quieres ejecutar. Sin embargo, podrían llegar a haber errores de ejecución y el bloque se dejará de ejecutarse.

- El bloque **except** se ejecutará cuando el bloque **try** falle debido a un error. Este bloque contiene sentencias que generalmente nos dan un contexto de lo que salió mal en el bloque **try**.

- Siempre deberías de mencionar el **tipo de error** que se espera, como una excepción dentro del bloque **except** dentro de **<tipo de error>** como lo muestra el ejemplo anterior.

- Podrías usar **except** sin especificar el **<tipo de error>.** Pero no es una práctica recomendable, ya que no estarás al tanto de los **tipos de errores** que puedan ocurrir.

Cuando se ejecute el código dentro del bloque try, existe la posibilidad de que ocurran diferentes errores.

- Por ejemplo, podrías acceder a una lista utilizando un índice fuera de rango, usar una clave incorrecta en un diccionario y tratar de abrir un archivo que no existe - todo esto dentro de un bloque **try**.

    - En este caso, podrías esperar lo siguiente: **IndexError, KeyError y FileNotFoundError.** Y tendrías que añadir un bloque **except** por cada tipo de error que puedas anticipar.

    - El bloque **else** se ejecutará solo si el bloque **try** se ejecuta sin errores. Esto puede ser útil cuando quieras continuar el código del bloque **try**. Por ejemplo, si abres un archivo en el bloque **try**, podrías leer su contenido dentro del bloque **else**.
    - El bloque **finally** siempre es ejecutado sin importar que pase en los otros bloques, esto puede ser útil cuando quieras liberar recursos después de la ejecución de un bloque de código, (**try, except o else).**

Los bloques else y finally son opcionales. En muchos casos puedes solo ocupar el bloque try para tratar de ejecutar algo y capturar los errores como excepciones en el bloque except.

COMO MANEJAR ZERODIVISIONERROR EN PYTHON

Considera la siguiente **función, dividir(),** toma 2 argumentos - **num y div** - y retorna el resultado o cociente de dividir **num/ div.**

CUANDO SE DIVIDE POR CERO DARÁ UN ERROR

Podemos tratar esta división entre cero como una excepción, haciendo lo siguiente:

- Desde el bloque **try** llama a la **función dividir().**

- En el bloque **except** tendremos una excepción en caso de que **div** sea igual a cero.

- En este ejemplo se hace una excepción a **ZeroDivisionError** (el tipo de error) el cual se especifica en **except**

- Cuando se detecte el error **ZeroDivisionError** se ejecutará el bloque **except** donde pondremos un mensaje informando que se trató de dividir entre cero.

Como enfrentar TypeError en Python

En esta sección, verás como usar **try** y **except** para manejar el error **TypeError** en Python. Una función llamada **mas_10(),** que toma un número como argumento, le añade 10, y retorna el resultado de la suma.

- Teniendo la variable **mi_num**, llamamos a la función **mas_10()** con **mi_num** como argumento, si el argumento es un tipo de dato válido, no se activará la excepción.
- De lo contrario, el bloque **except** con el tipo de error **TypeError** se activará, notificándole al usuario que el argumento es un tipo de dato inválido.

Como manejar IndexError en Python

Esto ocurre, ya que a veces es difícil estar al tanto de todos los cambios en un iterable y podrías estar usando un índice no válido para acceder a un elemento del iterable.
mira la siguiente lista:

```
mi_lista = ["Python","C","C++","JavaScript"]
```

En este ejemplo, mi_lista tiene 4 elementos. Los índices válidos son **0, 1, 2, 3** y **-1, -2, -3, -4** si usas indexación negativa.

```
mi_lista = ["Python","C","C++","JavaScript"]
print(my_lista[2])

# Salida
C++
```

Ya que 2es un índice válido, al imprimir **my_lista[2]** el elemento **c++** es mostrado en pantalla. Si tratas de acceder a la lista con un índice fuera del rango permitido, ocurrirá un error en la ejecución y aparecerá el error **IndexError:**

```
print(mi_lista[4])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-437bc6501dea> in <module>()
      1 mi_lista = ["Python","C","C++","JavaScript"]
----> 2 print(mi_lista[4])

IndexError: list index out of range
```

```
IndexError
Esta linea se puede traducir como:
Error de indice

list index out of range
Esta linea se puede traducir como:
indice de la lista fuera de rango
```

Podemos usar los bloques **try** y **except** para evitar el error anterior.
En el siguiente código, estas intentando acceder a un elemento con la variable **buscar_ind**

```
mi_lista = ["Python","C","C++","JavaScript"]
buscar_ind = 3
try:
  print(mi_lista[buscar_ind])
except IndexError:
  print("Lo siento, el indice esta fuera de rango")
```

```
mi_lista = ["Python","C","C++","JavaScript"]
buscar_ind = 3
try:
  print(mi_lista[buscar_ind])
except IndexError:
  print("Lo siento, el indice esta fuera de rango")
```

Aquí, **buscar_ind (3)** es un índice válido, y elemento en ese índice es impreso en pantalla:

```
JavaScript
```

Si **buscar_ind** está fuera del rango permitido, el bloque **except** reconoce el error **IndexError** como una **excepcion** y se mostrara nuestro pequeño mensaje en lugar de la descripción larga del error?.

```
mi_lista = ["Python","C","C++","JavaScript"]
buscar_ind = 4
try:
  print(mi_lista[buscar_ind])
except IndexError:
  print("Lo siento, el indice esta fuera de rango")
```

Como manejar FileNotFoundError en Python

Un error común al trabajar con archivos en Python es el error **FileNotFoundError.**

En el siguiente ejemplo, trataras de abrir el archivo **mi_archivo.txt** especificando su ruta en la función **open(),** después intentaras leerlo e imprimir su contenido.
Sin embargo, aún no has creado el archivo en la ruta especificada.
Si intentas correr el código siguiente, obtendrás el error **FileNotFoundError:**

```
mi_archivo = open("Contenido/datos_muestra/mi_archivo.txt")
contenido = mi_archivo.read()
print(contenido)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-4-4873cac1b11a> in <module>()
----> 1 my_file = open("my_file.txt")

FileNotFoundError: [Errno 2] No such file or directory: 'my_file.txt'
```

Usando try y except, puedes hacer lo siguiente:

- Tratar de abrir el archivo en el bloque **try**.
- En el bloque **except** tendremos una excepción en caso de que el archivo no exista y le notificaremos al usuario.
- Si el bloque **try** no tiene errores y el archivo si existe, leeremos e imprimiremos el contenido del archivo.
- En el bloque **finally**, cerramos el archivo para evitar desperdiciar recursos. Recuerda que el archivo será cerrado independientemente de lo que ocurra en los pasos de apertura y lectura del archivo

```
try:
  mi_archivo = open("Contenido/datos_muestra/mi_archivo.txt")
except FileNotFoundError:
  print(f"Lo siento, el archivo no existe")
else:
  contenido = mi_archivo.read()
  print(contenido)
finally:
  mi_archivo.close()
```

Ahora manejamos el error como una excepción y el programa termina solo con un pequeño mensaje:

```
Lo siento, el archivo no existe
```