

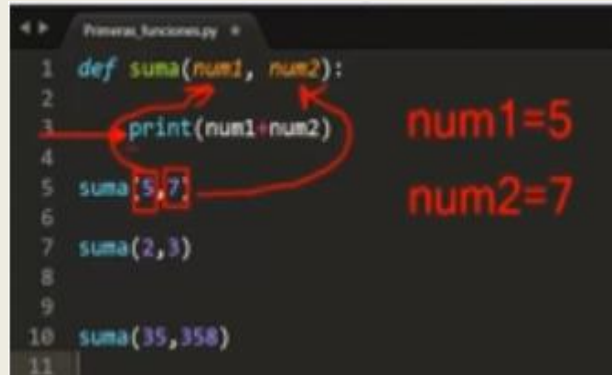
FUNCIONES

Parámetros

Hay algunas funciones **predefinidas** por el lenguaje Python, por ejemplo `print()` que hemos estado utilizando. La función está escrita en Python en los módulos de código que vienen al instalar el lenguaje.

Y las funciones **propias** que son las que nosotros escribimos y las que estamos aprendiendo a crear.

Recordamos que Python pasa los **parámetros** por **referencia**



```
1 def suma(num1, num2):
2
3     print(num1+num2)
4
5 suma(5,7)
6
7 suma(2,3)
8
9
10 suma(35,358)
11
```

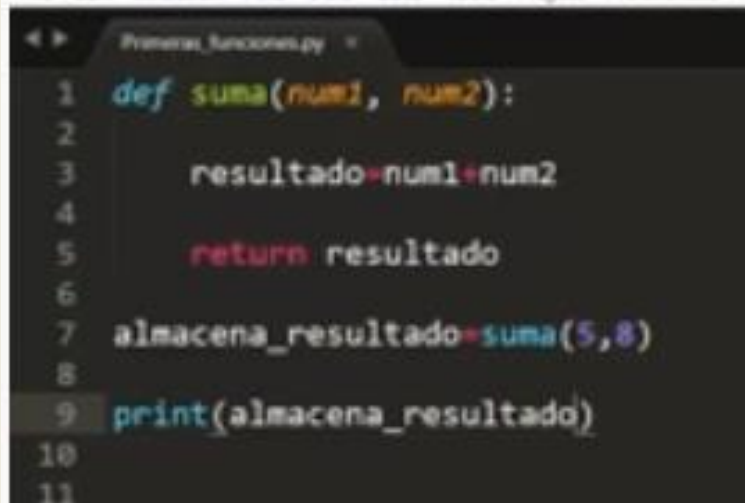
The image shows a code editor window titled 'Primeras funciones.py'. It contains a Python function definition `def suma(num1, num2):` followed by a `print(num1+num2)` statement. Below the function definition, there are three function calls: `suma(5,7)`, `suma(2,3)`, and `suma(35,358)`. Red arrows indicate the flow of data: one arrow points from the `num1` parameter to the value `5` in the first call, and another points from the `num2` parameter to the value `7`. To the right of the code, the text `num1=5` and `num2=7` is displayed in red, indicating the state of the variables during the first function call.

La primera vez que la llamamos `num1` tomará el valor 5 y `num2` el valor 7. Cuando la llamamos por segunda vez tomará los valores 2 y 3 y así cada vez que la llamemos.

Los parámetros que una función serán utilizados dentro de su algoritmo, a modo de **variables de ámbito local**. Si queremos utilizar valores generados dentro de la función en el programa que la llamó debemos pasar ese valor, *retornarlo*

Sentencia return

La sentencia *return* es opcional, puede devolver, o no, un valor y es posible que aparezca más de una vez dentro de una misma función.



```
Primeras_funciones.py
1 def suma(num1, num2):
2
3     resultado=num1+num2
4
5     return resultado
6
7 almacena_resultado=suma(5,8)
8
9 print(almacena_resultado)
10
11
```

return hace que termine la ejecución de la función cuando aparece y el programa continúa por su flujo normal, el flujo del programa continúa por la instrucción que sigue a la llamada de dicha función.

return que no devuelve ningún valor

La siguiente función muestra por pantalla el cuadrado de un número solo si este es par:

```
def cuadrado_de_par(numero):  
    if not numero % 2 == 0:  
        return  
    else:  
        print(numero ** 2)  
  
cuadrado_de_par(8)  
cuadrado_de_par(3)
```

Varios return en una misma función

La función `es_par()` devuelve `True` si un número es par y `False` en caso contrario

```
def es_par(numero):  
    if numero % 2 == 0:  
        return True  
    else:  
        return False  
  
es_par(2)  
es_par(5)
```

Devolver más de un valor con return

En Python, es posible devolver más de un valor con una sola sentencia return. Por defecto, con return se puede devolver una tupla de valores.

función `cuadrado_y_cubo()` que devuelve el cuadrado y el cubo de un número:

```
def cuadrado_y_cubo(numero):  
    return numero ** 2, numero ** 3  
  
cuad, cubo = cuadrado_y_cubo(4)  
|
```

Cada uno de nuestros archivos .py se denominan **módulos**. Estos módulos, a la vez, pueden formar parte de **paquetes**. Un paquete, es una carpeta que contiene archivos .py.

Como vimos en la clase anterior si la definición de la función se encuentra en un archivo .py y la llamada a la función en otro archivo .py debemos importar la función utilizando la sentencia `import`

El contenido de cada módulo, podrá ser utilizado a la vez, por otros módulos. Para ello, es necesario **importar los módulos** que se quieran utilizar. Para importar un módulo, se utiliza la instrucción `import`, seguida del nombre del paquete (si aplica) más el nombre del módulo (sin el .py) que se desee importar.

```
# -*- coding: utf-8 -*-  
import modulo           # importar un módulo que no pertenece a un paquete  
import paquete.modulo1 # importar un módulo que está dentro de un paquete  
import paquete.subpaquete.modulo1
```

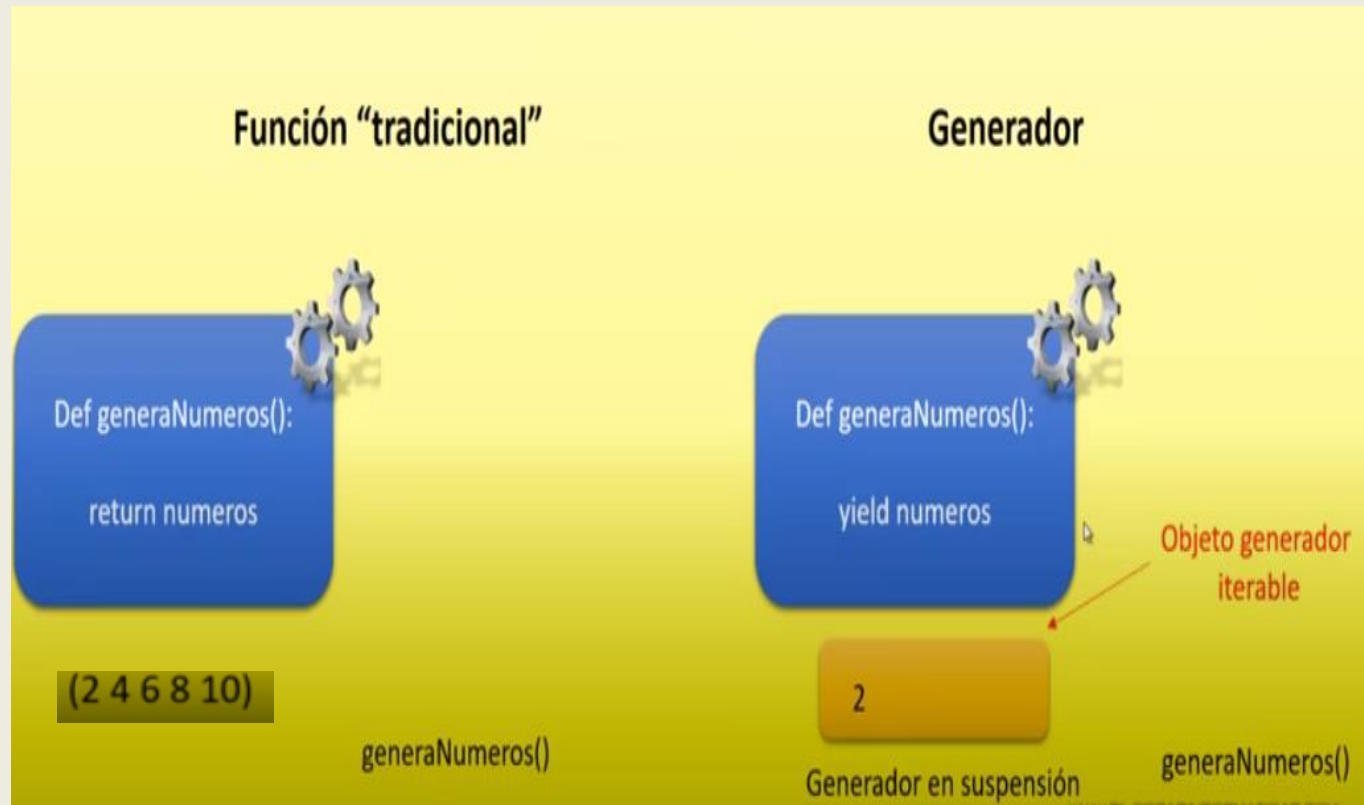
La instrucción `import` seguida de nombre_del_paquete.nombre_del_módulo, nos permitirá hacer uso de todo el código que dicho módulo contenga.

Nota Python tiene sus propios módulos, los cuales forman parte de su librería de módulos estándar, que también pueden ser importados.

Generadores

- Estructuras que extraen valores de una función y se almacenan en objetos iterables (que se pueden recorrer)
- Estos valores se almacenan de uno en uno.
- Cada vez que un generador almacena un valor, esta permanece en un estado pausado hasta que se solicita el siguiente. Esta característica es conocida como “suspensión de estado”

Función para crear lista de números pares



En la función tradicional construye toda la lista y nos devuelve toda la lista

En un generador nos devuelve un objeto generador iterable el cual está almacenado el primer valor que nos ha de devolver luego el generador pasa a un estado de suspensión y el control pasa al programa que realizó esa llamada. Si volvemos a llamar a ese generador genera el segundo valor y vuelve a entrar en suspensión, y así sucesivamente.

La diferencia fundamental es que nos va devolviendo los valores de a uno en un objeto generador iterable.

En la función tradicional uso **return** en el generador **yield**

¿Y esto para qué?

Ventajas de los Generadores

- Son más eficientes que las funciones tradicionales
- Muy útiles con listas de valores infinitos
- Bajo determinados escenarios, será muy útil que un generador devuelva los valores de uno en uno

Si queremos generar una lista de números pares

```
def generaPares(Limite):  
    num=1  
    miLista=[]  
    while num<limite:  
        miLista.append(num*2)  
        num=num+1  
    return miLista
```

```
def generaPares(Limite):  
    num=1  
    while num<limite:  
        yield num*2  
        num=num+1  
  
print(generaPares(10))
```

Yield arma el objeto iterable.

Si no quiero que me devuelva todos los valores sino de a uno
Nos irá mostrando valor a valor

Podemos ver que cada vez que llamamos al generador muestra el siguiente valor

```
def generaPares(limite):  
    num=1  
    while num<limite:  
        yield num*2  
        num=num+1  
devuelvePares=generaPares(10)  
print(next(devuelvePares))
```

Método next()

```
devuelvePares=generaPares(10)  
  
print(next(devuelvePares))  
print("Aquí podría ir más código...")  
print(next(devuelvePares))  
print("Aquí podría ir más código...")  
print(next(devuelvePares))
```

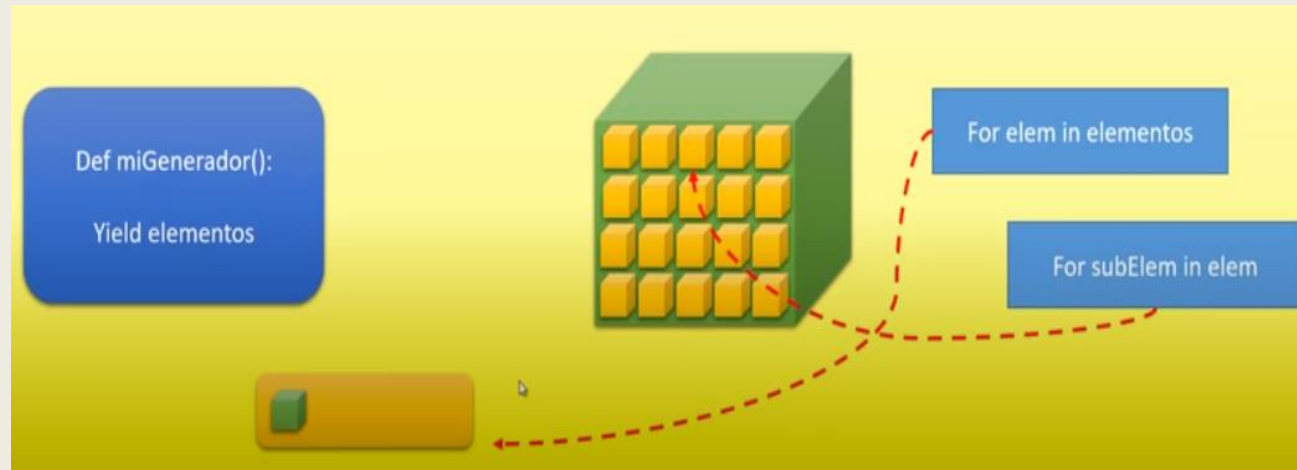
El control del flujo pasa constantemente del objeto generador al programa que lo llama. Entre llamada y llamada entra en un estado de suspensión. Puedo crear una función tradicional pero no será tan eficiente en la utilización de recursos porque tiene que generar todos los elementos. En cambio el generador los construye a medida que lo llamo.

Instrucción yield from

Utilidad: simplifica el código de los generadores en caso de tener que usar bucles anidados

Necesito entrar a cada uno de los elementos que me devuelve el generador, estructura similar a un array de dos dimensiones debería usar for anidados para acceder a cada elemento.

Python nos ofrece una simplificación



```
def devuelve_ciudades(*ciudades):  
    for elemento in ciudades:  
        yield elemento  
  
ciudades_devueltas=devuelve_ciudades("Madrid", "Barcelona", "Bilbao", "Valencia")  
  
print(next(ciudades_devueltas))
```

* le estamos diciendo que va a recibir un número indeterminado de elementos y además que los va a recibir en forma de tupla
next va recorriendo el objeto iterable que me devuelve

El print mostrará cada string (ciudad)

Si quiero recorrer cada elemento en forma individual debería utilizar otro for
en lugar de esto puedo usar yield from

Es decir acceder a cada letra de la palabra

```
def devuelve_ciudades(*ciudades):  
    for elemento in ciudades:  
        for subElemento in elemento:  
            yield subElemento  
            |  
  
ciudades_devueltas=devuelve_ciudades("Madrid", "Ba  
print(next(ciudades_devueltas))  
print(next(ciudades_devueltas))
```

```
def devuelve_ciudades(*ciudades):  
    for elemento in ciudades:  
        #for subElemento in elemento:  
            yield from elemento  
            +  
  
ciudades_devueltas=devuelve_ciudades("Madri  
print(next(ciudades_devueltas))  
print(next(ciudades_devueltas))
```

Va a recorrer cada letra de cada string.

El field from construye un objeto iterable dentro del elemento iterables

EXCEPCIONES

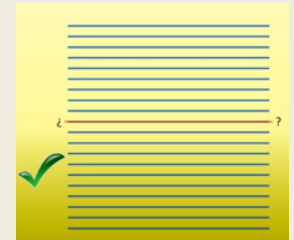
¿Qué son? ¿Cómo se manejan?



Las excepciones son errores que ocurren durante la ejecución del programa. La sintaxis del código es correcta pero durante la ejecución ha ocurrido “algo inesperado”.

Cuando ocurre algo así la ejecución se detiene y el resto del programa no se ejecuta

Este tipo de errores de ejecución se pueden controlar para que la ejecución del programa continúe. Es lo que se conoce como manejo o control de excepciones.



```
Traceback (most recent call last):
  File "prueba_excepciones.py", line 29, in <module>
    print(divide(op1,op2))
  File "prueba_excepciones.py", line 11, in divide
    return num1/num2
ZeroDivisionError: division by zero
```

← Pila de llamadas

Cuando ha ocurrido el error esta información que nos muestra Python es muy valiosa.

Primero está la pila de llamadas que nos informa que ha ido ocurriendo en qué línea está el error, en este archivo, en qué función y en qué línea está el error.

Abajo también nos da el nombre del error que ha ocurrido.


```
num1=float(input("Ingrese número 1: "))
num2=float(input("Ingrese número 2: "))
try:
    print("El resultado es ", num1/num2)
except:
    print("No se puede ejecutar")
```

Si queremos controlar esta circunstancia.

La instrucción susceptible del error la tenemos que poner dentro de una instrucción try:... except:

Si se produce un error deja de ejecutar las instrucciones que hay en try y va a las que hay en except

En try podemos la instrucción y en except el nombre del error.

```
try:
    return num1/num2

except ZeroDivisionError:
    print("No se puede dividir entre 0")
    return "Operación errónea"
```

Si el error que se ha producido coincide con el error del except ejecuta las líneas que se encuentran allí. Si el error que está aconteciendo no es la que está programado el programa se interrumpe.

```
while True:
    try:
        op1=(int(input("Introduce el primer número: ")))
        op2=(int(input("Introduce el segundo número: ")))
        break
    except ValueError:
        print("Los valores introducidos no son correctos.")
```

Este código genera un **bucle** y tanto si op1 u op2 son incorrectos va a ejecutar el except y en caso de ser correcto termina el bucle con la opción **break**

Capturar varias excepciones

```
try:
    op1=(float(input("Introduce el primer número: ")))
    op2=(float(input("Introduce el segundo número: ")))
    print("La división es: " + str(op1/op2))
except ValueError:
    print("El valor introducido es erróneo")
except ZeroDivisionError:
    print("No se puede dividir entre 0!")
```

Va a ejecutar el except según el error que se produzca

Si tuviéramos muchas opciones de errores a considerar y no quisiéramos poner detallada una por una, podríamos hacer una captura de error general
Podemos poner el except: sin ninguna especificación del error. Entonces todos los errores pasaran por ahí.

Claúsula Finally

Dentro del try: except: también tenemos otra opción que es finally: las líneas de código que hay allí se ejecutarán tanto si es correcto o si hay error. Es decir, se ejecutarán siempre.

Por lo que sino pusiera except: y ocurriera un error el finally se ejecutará igual, y el programa no se abortará.

```
try:
    op1=(float(input("Introduce el primer número: ")))
    op2=(float(input("Introduce el segundo número: ")))
    print("La división es: " + str(op1/op2))
except ValueError:
    print("El valor introducido es erróneo")
except ZeroDivisionError:
    print("No se puede dividir entre 0!")
finally:
    print("Cálculo finalizado")
```

```
num1=float(input("Ingrese número 1: "))
num2=float(input("Ingrese número 2: "))
try:
    print("El resultado es ", num1/num2)
except:
    pass
```

Si en except no quiero que ejecute nada puedo poner pass

Creación de excepciones propias

La sentencia **raise** permite al programador forzar a que ocurra una excepción específica.

```
x = 10
if x > 5:
    raise Exception('x should not exceed 5. The value of x was: {}'.format(x))
```

Raise nos permite definir el error y personalizar el mensaje