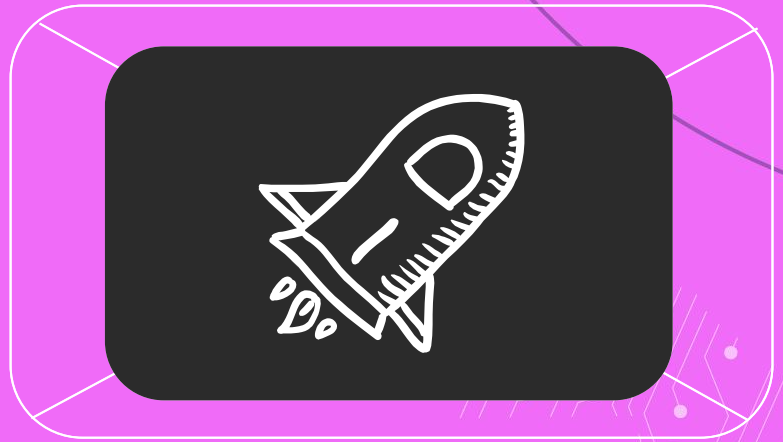
Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

Semana 4
¡Bienvenidos!



Temas de hoy

1. Introducción a la programación funcional
2. Definición y uso de funciones
3. Parámetros y retorno de valores
4. Alcance de variables

1

Introducción a la programación funcional

Hasta el momento veníamos viendo programación imperativa, con la cual, damos instrucciones de manera secuencial (una línea de código trás otra) a nuestro programa.

En este tipo de programación, la idea es cambiar el estado de un programa, mediante la modificación de variables.

La programación funcional es un paradigma de programación que se enfoca en el uso de funciones como bloques constructores fundamentales de un programa. La programación funcional se centra en **qué** queremos calcular, más que en **cómo** hacerlo 🧐

2 Funciones

Una función es un bloque de código diseñado para realizar una tarea específica. Una función toma una entrada, realiza una operación y puede devolver un valor. Nos ayuda a evitar la repetición de código y a descomponer programas complejos en tareas más simples y manejables 🤗

Funciones: Crear desde cero

Comenzamos con el nuevo paradigma

Como mencionamos anteriormente, las funciones son un bloque de código que nos da la posibilidad de reutilizarlo en cualquier parte de nuestro programa.
Para definir una función debemos darle un nombre a nuestra función (muy similar a lo que hacíamos con nuestro encabezado en Pseudocódigo 😉). Para ello ocuparemos la palabra reservada **def** seguido de un nombre descriptivo acorde a la tarea que resolverá la función y finalizamos la sintaxis agregando ambos paréntesis y dos puntos como podemos observar en el ejemplo.

Dentro de este bloque estaremos desarrollando el algoritmo y casi siempre al final y de forma opcional, se coloca una sentencia de retorno que es el encargado de que la función logre devolver un valor deseado: para esto usamos la palabra reservada **return**.

```
1  def saludar():
2      return "Hola Mundo!"
3
4  print(saludar())
```

```
Hola Mundo!
```

Siguiendo con el ejemplo, una vez creada la función bajo el nombre de *saludar()* podremos utilizarla en cualquier parte de nuestro código, como por ejemplo en la línea 4 estaremos invocando la función *saludar()* que lo que hará es retornar el mensaje “Hola Mundo!” y luego será mostrado en pantalla debido al print() como podemos ver en la segunda imagen.

Una función en Python siempre devolverá un valor. Aunque no se declare un “return”, la función devolverá el valor predeterminado de “None”

Nuevo paradigma, más beneficios

Principios clave de la programación funcional

Inmutabilidad: Los datos no se modifican después de ser creados. En su lugar, se crean nuevas versiones de los datos.
Funciones puras: Las funciones no tienen efectos secundarios, es decir, no modifican el estado externo.
Composición de funciones: Las funciones pueden ser combinadas para crear funciones más complejas.
Recursión: Se utiliza en lugar de bucles para repetir operaciones.
Funciones de alto orden: Pueden tomar funciones como argumentos o devolver funciones como resultado.

¿Por qué usar programación funcional?

Ahora que comenzamos a cambiar el paradigma, veremos que los códigos se pueden mejorar aún más, y el uso de la programación funcional nos ayuda:

Código más limpio y conciso: Al evitar efectos secundarios, el código es más fácil de entender y mantener.
Mayor seguridad: La inmutabilidad reduce la posibilidad de errores causados por modificaciones accidentales de datos.
Fácil de probar: Las funciones puras son más fáciles de probar de forma aislada.
Paralelismo y concurrencia: La programación funcional se adapta bien a la ejecución en paralelo, ya que las funciones puras no comparten estado.

3

Parámetros y Retorno

Basándonos en la función anterior, habrás notado que la función era muy simple, no hace más que imprimir un valor en la terminal.

¿Qué pasa si queremos añadir datos a la función?

Para ello utilizaremos **parámetros y argumentos** 😌

Funciones: Parámetros

Es momento de ingresar valores

Los parámetros son variables **momentáneas** y **locales** que declaramos cuando creamos una función para que pueda recibir valores cuando la función es llamada.
Los parámetros son declarados entre los dos paréntesis separados por una coma.

```
def nombre de la función(parámetros):
    bloque de código

#invocamos a la función
variable = función(argumentos)

#mostramos el resultado
print(variable)
```

```
funciones.py > ...
1  def sumar(a, b):
2      return a + b
3
4  resultado = sumar(3, 4)
5  print(f'El resultado es: {resultado}')

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINA

PS C:\Users\Pc\Documents\Info\Python> & C:/Users
Documents/Info/Python/funciones.py
El resultado es: 7
```

En el ejemplo podemos observar que declaramos un parámetro **a** y un parámetro **b** los cuales estarán recibiendo un valor cada uno para su uso en el algoritmo.
En esta función, cuando **a** y **b** tengan un valor, la función sumará ambos parámetros y retornará su resultado con un mensaje.

Funciones: Parámetros

Luego al momento de invocar la función estaremos obligados a pasarle la misma cantidad de argumentos que de parámetros para que funcione correctamente, en caso contrario, nos devolverá un error.

Esos valores que le damos al momento de invocar la función se llaman **argumentos.**

En el ejemplo, los argumentos son 3 y 4, por lo que **a** tomará el valor de 3 y, **b** el valor de 4 según el orden.
Y la función nos mostrará en pantalla el mensaje con el valor de **7** como se muestra en el retorno.

Los parámetros vistos este ejemplo se catalogan como parámetros posicionales, ya que los argumentos se asignan a sus parámetros según el orden en el que están entre los paréntesis. Por eso el primer argumento se asignaba al primer parámetro, y el segundo argumento con el segundo parámetro. Y en caso de haber más, seguiría ese orden sucesivamente.

Tipos de parámetros

Así como existen los parámetros posicionales, también existen los siguientes tipos de parámetros:

Valores por Defecto: Se asignan si no se proporciona un valor.

```
def saludar(nombre, mensaje="Hola"):
    return f"{mensaje}, {nombre}!"
```

Si a un parámetro le asignamos un valor al momento de declararlo, se volverá opcional el pasarle un argumento.
Por ejemplo, en este caso le asignamos el valor “Hola” al parámetro *mensaje*. Por lo tanto, al momento de invocar a la función, es obligatorio pasarle un argumento al parámetro *nombre* pero es opcional pasar un segundo argumento porque en caso de no recibir ningún argumento usará el “Hola” ya asignado.

Funciones: Parámetros

Como podemos verlo acá:

```
print(saludar("Mundo"))
Hola, Mundo!
```

Pero en el caso de que pasemos un segundo argumento, ocupará el lugar del "Hola" ya asignado predeterminadamente.

```
print(saludar("Mundo", "Buenos días"))
Buenos días, Mundo!
```

Parámetros *args, **kwargs

Los parámetros ***args** nos permiten pasar un número variable de argumentos posicionales. Si bien no es obligatorio utilizar la palabra "args" al ser una simple convención entre programadores, se recomienda su uso. A su vez, lo que sí es imprescindible para que el parámetro acepte una indefinida cantidad de argumentos es el asterisco "*****".

```
def receta_de_cocina(titulo, *args):
    print(f"Receta de {titulo}")
    print("Ingredientes:")

    for ingrediente in args:
        print(ingrediente)
```

La función *receta_de_cocina* debe recibir un *titulo* obligatoriamente y luego la cantidad de ingredientes que se desee. La función mostrará por consola de qué se trata la receta y listará sus ingredientes.

```
receta_de_cocina('Torta de limon', 'Harina', 'Leche', 'Limon')
```

Como podemos observar, los tres argumentos pasados luego del título (primer argumento), se guardarán en **forma de tupla** dentro del parámetro *args* el cual lo estaremos recorriendo con un **bucle for** como aprendimos ya en clases anteriores.

Funciones: Parámetros

La salida quedaría así:

```
Receta de Torta de limon
Ingredientes:
Harina
Leche
Limon
```

Por otro lado, los parámetros ****kwargs** nos permiten pasar un número variable de argumentos al igual que ***args** con la diferencia de que deben ser nombrados.
Al igual que el tipo de parámetro anterior, la palabra *kwargs* es una convención simplemente, lo indispensable para su uso son los DOS asteriscos "**".

```python
def suma(**kwargs):
    resultado = 0

    for clave, valor in kwargs.items():
        print(clave, "=", valor)
        resultado += valor

    return resultado
```

Al decir "argumentos nombrados" nos referimos que al momento de escribir los argumentos, los mismos deben tener un nombre que los identifique seguidos de un "=", como podemos observar acá:

```python
print( suma(a=3, b=10, c=3) )
```

La función *suma* nos permite sumar una cantidad indefinida de números que queramos.
Otra diferencia muy importante con respecto a **args* es que los argumentos se guardarán en el parámetro ***kwargs* en **forma de diccionario** donde el nombre del argumento será la clave con su valor correspondiente. Por lo tanto lo tendremos que manipular y recorrer con un bucle for como tal.

```
a = 3
b = 10
c = 3
16
```

Funciones: Retorno de valores

El valor de retorno es el resultado que una función devuelve después de completar su tarea y para ello utilizamos la palabra clave **return**.

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

print(es_par(4))
```

```
PROBLEMS  OUTPUT  PORTS  DEBUG CONSOL
True
```

En este caso la función *es_par()* nos devuelve *True* o *False* en caso de ser par o no.

Pero Python tiene la particularidad de que nos permite retornar múltiples valores desde una función utilizando tuplas.

```
def operaciones(a, b):
    suma = a + b
    resta = a - b

    return suma, resta

print(operaciones(5, 3))
```

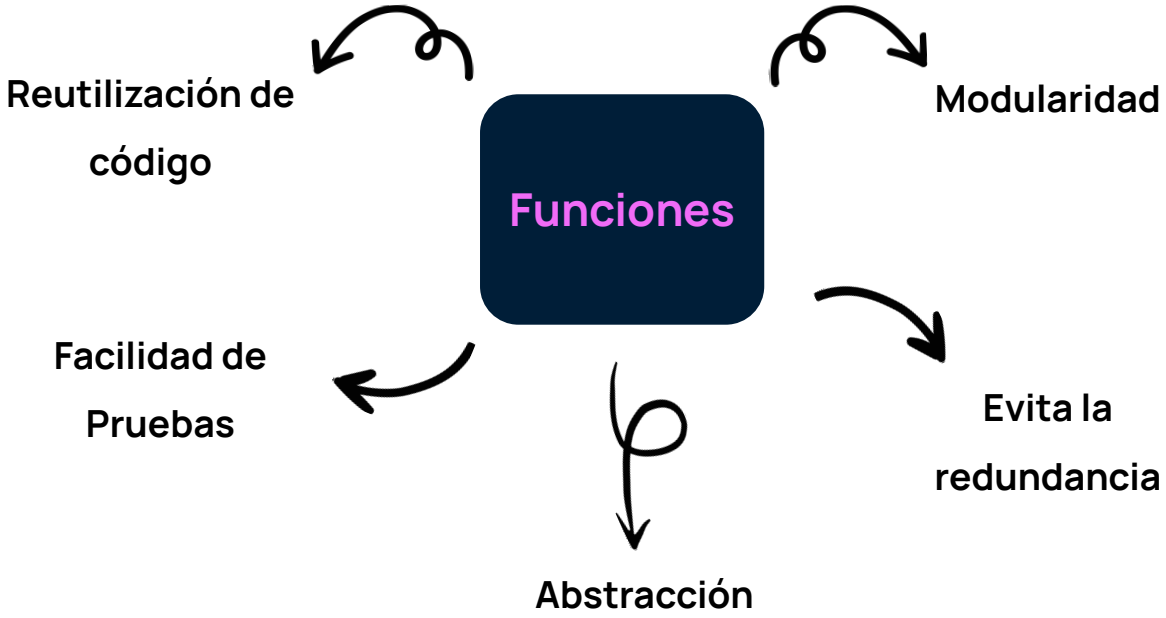
```
PROBLEMS  OUTPUT  PORTS  DEBUG CON
(8, 2)
```

Por ejemplo, *operaciones()* recibe 2 argumentos los cuales serán sumados y restados entre sí.
Si bien retornamos ambos resultados, podemos observar en la terminal que el retorno es una tupla con ambos resultados.
El hecho de que *operaciones()* nos devuelva una tupla, nos permite asignar los resultados del retorno de una forma más dinámica:

```
resultado_suma, resultado_resta = operaciones(5, 3)

print(f"El resultado de la suma es: {resultado_suma}")
print(f"El resultado de la resta es: {resultado_resta}")
```

```
El resultado de la suma es: 8
El resultado de la resta es: 2
```

Ventajas de utilizar funciones



Ventajas: Reutilización de código

- Como estuvimos mencionando en repetidas ocasiones, una función puede ser llamada múltiples veces en diferentes partes del programa sin necesidad de reescribir el mismo código, evitando ser redundante con el mismo cálculo.

```
1  def sumar(a, b):
2      return a + b
3
4  print(sumar(2, 3))
5  print(sumar(4, 5))
6  print(sumar(6, 8))
7  print(sumar(12, 55))
8  print(sumar(231, 534))
9  print(sumar(756564, 243156))
10
```

```
PROBLEMS  OUTPUT  PORTS  DEBUG CONSOLE

5
9
14
67
765
999720
```

- La función *sumar()* se reutiliza para realizar la misma operación en diferentes contextos.
  En caso contrario, sin funciones, deberíamos estar reescribiendo la operación cada vez que quisiéramos sumar.
  Cuanto más complejas las funciones, más sentido toma la reutilización de código.

Ventajas: Modularidad

- Las funciones permiten dividir un programa en módulos más pequeños y manejables, facilitando la organización y mantenimiento del código.

```python
def leer_datos():
    # código para leer datos
    pass

def procesar_datos():
    # código para procesar datos
    pass

def guardar_datos():
    # código para guardar datos
    pass

leer_datos()
procesar_datos()
guardar_datos()
```

- Cada función realiza una tarea en específico, haciendo que el código sea más fácil de entender a simple vista, y mucho más organizado.
  Su contraparte sería tener una sola función con todo el código de cada tarea, una seguida de la otra, lo que sería engorroso al momento de leer e interpretar.

Ventajas: Facilidad de Pruebas

- Las funciones al ser individuales pueden ser probadas de manera aislada, lo que nos facilita la detección y corrección de errores.

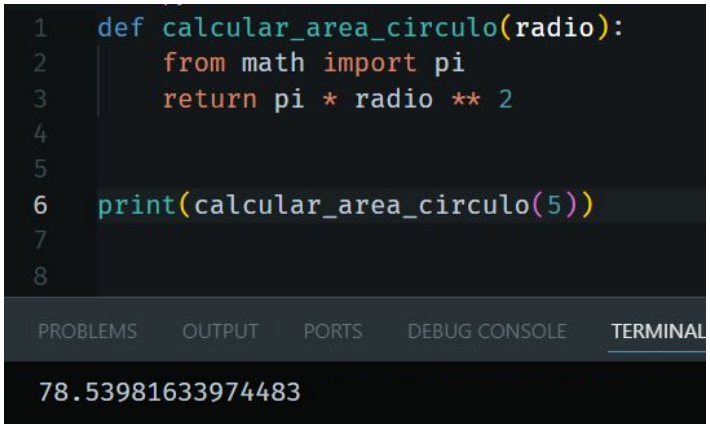
```python
def multiplicar(a, b):
    return a * b

# Prueba de la función
assert multiplicar(2, 3) == 6

assert multiplicar(0, 3) == 0
```

Probamos la función *multiplicar()* con diferentes entradas para asegurarnos de que funciona correctamente.
Utilizamos la palabra reservada **assert** que nos ayuda a verificar si una condición es verdadera. En caso de que sea falsa, nos retorna un error en la terminal

- En este caso la condición era de que la multiplicación entre 2 y 3 sea igual a 3, por lo tanto es falso y en la terminal no aparece un error.

Ventajas: Abstracción

- Las funciones permiten ocultar los detalles de la implementación y proporcionan una interfaz más sencilla para el usuario.



- La función oculta los detalles del cálculo del área y proporciona una forma más simple de obtener el resultado.
  Como sabemos que esta función puede ser reutilizada en cualquier parte de nuestro programa, al encapsular la operación en una función, estamos generando una forma más simple de poder obtener el área de un círculo, sin necesidad de que el usuario sepa cómo estamos generando al resultado.

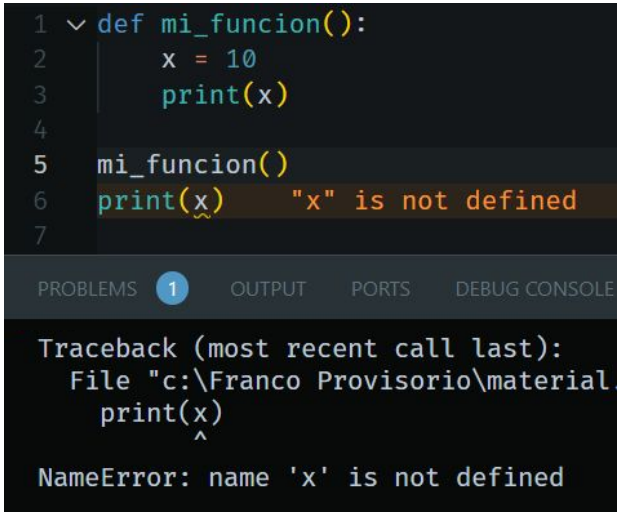
4

Alcance de las Variables

Por alcance de una variable nos referimos a la accesibilidad que tiene una variable en nuestro programa. Si la variable la creamos dentro de una función, ¿podemos utilizarla fuera de la misma? ¿ Y si quiero ocupar una misma variable en todo mi programa?

De eso dependerá el alcance de las variables locales y globales 🤩

Alcance de variables: Variables locales



Las variables de alcance local o **variables locales**, son todas las variables definidas en el interior de una función que al momento de culminar la ejecución de dicha función, serán borradas de la memoria y no podremos cambiarlas desde fuera de la función, es decir, que quedan fuera de alcance.

En este caso, la variable **x** es definida dentro de la función, y la misma es **ejecutada y finaliza** en la línea 5.

Por lo tanto al momento de querer hacer un print de **x** en la línea 6, nos devuelve error en la terminal ya que **x** dejó de existir en memoria. Solamente puede ser usada dentro de la función.

Alcance de variables: Variables globales

Las variables de alcance global o **variables globales**, son todas las variables definidas por fuera de una función, por lo tanto serán accesibles por cualquier función siempre y cuando estén declaradas previamente a la misma.

¡Importantísimo! A pesar de que las funciones puedan acceder a las variables, hay que tener en cuenta que la función no podrá modificar el valor de dichas variables.

En este caso como podemos observar, la variable **x** está siendo definida antes que la función y por fuera de la misma. Por lo tanto la función puede acceder a ella para ejecutar el print y además podemos realizar un print también por fuera de la función luego de que sea finalizada su ejecución.

Conclusión final

La programación funcional es un fundamento esencial en la programación y nos ayuda a escribir código eficiente, modular y fácil de entender y mantener.

Dominar estos conceptos es vital para cualquier desarrollador, el uso correcto de funciones, parámetros y el entendimiento del alcance de variables mejora la eficiencia y mantenibilidad del código exponencialmente.

Con mucha práctica y aplicación constante, estos principios se convertirán en herramientas poderosas en el arsenal de cualquier programador 🐱💻

Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

¡Nos vemos En la próxima clase!