TrabajoInvestigacion

February 3, 2025

Prof. Andrés Mena Abarca # Exploración Teórica y Aplicación Práctica de las Funciones en Python Planteamiento del Problema Las funciones en Python son una herramienta esencial en la programación estructurada y funcional. Su correcto uso permite mejorar la reutilización del código, modularización y escalabilidad de los programas. Sin embargo, muchos estudiantes desconocen las mejores prácticas en su implementación y el impacto en la eficiencia del código. Pregunta de investigación:

0.1 ¿Cómo influyen las funciones en Python en la modularización y eficiencia de los programas en desarrollo de software?

Ventajas de la modularidad en Python

Organización del código: Al modularizar el código en Python, se divide en diferentes módulos, lo que facilita la organización. Cada módulo puede contener un conjunto de funciones relacionadas o una parte específica de la lógica del programa. Esto hace que sea más fácil para los desarrolladores entender y trabajar en diferentes partes del código.

Reutilización de código: Modularizar el código en Python también permite la reutilización de código. Cuando se crea una función o un conjunto de funciones en un módulo, se pueden importar y utilizar en otros lugares de nuestro código o incluso en otros proyectos. Esto ahorra tiempo y esfuerzo, ya que no es necesario volver a escribir el mismo código una y otra vez.

Colaboración efectiva: En proyectos de desarrollo de software en equipo, modularizar el código en Python es fundamental. Cada miembro del equipo puede trabajar en módulos específicos sin afectar al trabajo de los demás. Esto facilita la colaboración y acelera el proceso de desarrollo.

1 matematicas.py

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

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

import matematicas

resultado = matematicas.suma(5, 3) print(resultado) # Salida: 8

Desarrollo del Notebook Crear un notebook en Jupyter siguiendo la siguiente estructura. Cada sección debe contener una combinación de texto explicativo (Markdown) y código en Python (Code Cells).

1.1 Incluir una breve introducción sobre el propósito del notebook y código.

Jupyter Notebook es una herramienta web basada en celdas que permite programar código en Python1. Es un entorno de desarrollo interactivo con live code, que muestra la ejecución del código a través del navegador web2. Jupyter Notebook es una aplicación web de código abierto que permite a los usuarios crear y compartir documentos que contienen código en vivo, ecuaciones, visualizaciones y texto narrativo

2 Explicar la importancia del uso de funciones en Python.

Las funciones en Python permiten encapsular código reutilizable, mejorando la modularización y eficiencia en el desarrollo de software.

2.1 3.1 Definición y Propósito de las Funciones en Python

2.2 ¿Qué son las funciones?

Las funciones en Python permiten encapsular código reutilizable, mejorando la modularización y eficiencia en el desarrollo de software. En términos simples, una función en Python es un bloque de código reutilizable que realiza una tarea específica. Imagine una función como una caja negra usted introduce ciertos datos (argumentos) y la función procesa estos datos para producir un resultado (valor de retorno). Las funciones permiten dividir un programa complejo en partes más pequeñas y manejables, promoviendo la modularidad y la legibilidad. `def` seguida del nombre de la función, paréntesis que pueden contener parámetros, y dos puntos ( ) para indicar el inicio del bloque de código de la función; La estructura básica de una función se muestra a continuación

python def nombre_de_la_funcion(parámetro1, parámetro2, …) # Código de la función return valor_de_retorno

[1]:
```
def saludar():
    print("¡Hola, bienvenido a Python!")

saludar()
```

```
¡Hola, bienvenido a Python!
```

2.3 Beneficios de modularizar código con funciones.

Modularidad Las funciones dividen el código en unidades independientes, lo que facilita la organización y el mantenimiento del código. Reutilización de código Las funciones se pueden llamar desde diferentes partes del programa, evitando la duplicación de código. Eficiencia La reutilización de código reduce el tamaño del programa y mejora la eficiencia. Legibilidad Las funciones hacen que el código sea más fácil de leer y comprender. Depuración Las funciones facilitan la identificación y corrección de errores, ya que se puede probar cada función de forma independiente. Pruebas Las funciones se pueden probar de forma independiente, lo que facilita la verificación de su correcto funcionamiento. Documentación Las funciones se pueden documentar de forma clara y concisa, lo que facilita su comprensión y uso.

2.4 Importancia de la reutilización del código.

Se pueden llamar desde diferentes partes del programa, evitando la duplicación de código.

3 3.2 Tipos de Funciones en Python

3.1 Incluir una descripción y al menos un ejemplo de código para cada tipo de función:

3.2 Funciones con y sin retorno.

Las funciones sin retorno son aquellas que no devuelven ningún valor explícito. Estas funciones realizan una tarea específica y luego terminan su ejecución.

[ ]:
```python
#ejemplos sin retorno
def saludar(nombre):
    print(f"¡Hola, {nombre}!")

# Llamada a la función
saludar("Ana") #función sin retorno
```

En Python, se utiliza la instrucción return para especificar qué valor debe ser devuelto por la función. Aquí tienes un ejemplo sencillo de una función que calcula la suma de dos números: En este caso, la función suma toma dos números como argumentos, los suma y luego devuelve el resultado utilizando la instrucción return.

[ ]:
```python
def suma_varios(*args):
      total = 0
      for num in args:
            total += num
      return total
```

[ ]:
```python
def calcular_saldo(cuenta, transacciones):
      saldo = 0
      for transaccion in transacciones:
            saldo += transaccion
      return saldo
```

3.3 Funciones con parámetros y valores predeterminados.

Esta es y una función con parametros y valor predeterminiado al poner el Igual si no lo da el usuario lo toma por default

[9]:
```python
def oper(x, y, op='+'): # esta es y una función con parametros y valor predeterminiado al poner el Igual si no lo da el usuario lo toma por default
    if op == '+':
        return x + y
    elif op == '-':
        return x - y
```

```
    elif op == '/':
        return x / y
    else:
        return None
resultado = oper(5, 6, '-') # dando operador
print(f'El resultado de la operacion es {resultado}')
resultado = oper(5, 6) #sin dar operador hace suma
print(f'El resultado de la operacion es {resultado}')
```

```
El resultado de la operacion es -1
El resultado de la operacion es 11
```

3.3.1 Funciones con Uso de *args y **kwargs.

Empaquetado de tuplas de argumento *arg Cuando un nombre de parámetro en una definición de función de Python va precedido de un asterisco (), indica el empaquetado de tuplas de argumentos. Los argumentos correspondientes de la llamada de función se empaquetan en una tupla a la que la función puede hacer referencia mediante el nombre de parámetro especificado. Este es un ejemplo: *args vuelve una tupla de longitud o largo variable por ejemplo de avg promedio no sabemos cuantos son los números que vamos a promediar por eso lo podemos poner en una tupla con args y ahí los podemos sumar con for o con una función de sum(args) / len(args)

[4]:
```
def f(*args):  # el asterisco en una función es que el parametro es una tupla␣
 ↪cuando viene ** dos astericos el parametro es un dicionario
    print(args)
    print(type(args), len(args))
    for x in args:
        print(x)
f(1, 2, 3)
#************************************
def avg(*args):  # es una tupla que recorreremos para totalizar y dividir por␣
 ↪la cantidad de elementos y asi sacar el promedio
    total = 0
    for i in args:
        total += i
        return total / len(args)
promedio = avg(1, 2, 3, 4, 5) # devuelve 3.0
print(f'El promedio es {promedio}')
 #********
def PromedioTupla(*args):                  # una forma mas fácil de hacer la␣
 ↪misma operación de sumar tuplas
    return sum(args) / len(args)
promedio = PromedioTupla(1, 2, 3, 4, 5 , 6, 7, 8, 9, 10) # devuelve 3.0
print(f'El promedio de tupla con funcion sum es {promedio}')
```

```
(1, 2, 3)
<class 'tuple'> 3
1
```

```
2
3
El promedio es 0.2
El promedio de tupla con funcion sum 5.5
```

3.4 funciones con **kwargs

Una función con doble asterisco al inicio es que es un diccionario no se sabe con cuantas claves y valores entonces es de longitud o largo variable. **kwargs es como decir Key y Worth args === osea Llave y Valor como en un Diccionario {} es abierto porque no sabemos cuantas llaves y valores vamos a tener se maneja igual a como manejamos el diccionario de Python, siendo un diccionario podemos trabajar con while o for para recorrerlo o usar funciones de diccionario

[6]:
```
def funcionDiccionario(**kwargs):
    print(kwargs)
    print(type(kwargs))
    for key, val in kwargs.items():
        print(key, '->', val)
funcionDiccionario(Estudiante='JOSE FRANCISCO', CEDULA='401580001',␣
 ↪Dirección='Heredia')
```

```
{'Estudiante': 'JOSE FRANCISCO', 'CEDULA': '401580001', 'Dirección': 'Heredia'}
<class 'dict'>
Estudiante -> JOSE FRANCISCO
CEDULA -> 401580001
Dirección -> Heredia
```

Los tres (parámetros posicionales estándar, *args y **kwargs) se pueden utilizar en una definición de función de Python. Si es así, entonces deben especificarse en ese orden:

[7]:
```
def f(a, b, *args, **kwargs):
    print(F'a = {a}')
    print(F'b = {b}')
    print(F'args = {args}')
    print(F'kwargs = {kwargs}')
f(1, 2, 'foo', 'bar', 'baz', 'qux', x=100, y=200, z=300)
```

```
a = 1
b = 2
args = ('foo', 'bar', 'baz', 'qux')
kwargs = {'x': 100, 'y': 200, 'z': 300}
```

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

[28]:
```
suma = lambda a, b: a + b
print(suma(2,4))
```

```
6
```

5 Funciones recursivas.

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. 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.

[29]:
```
def factorial_normal(n):
    r = 1
    i = 2
    while i <= n:
        r *= i
        i += 1
    return r

factorial_normal(5) # 120
```

[29]:
```
120
```

6 Generadores (yield) siguiente elemento.

Hay veces que es preferible que una función vaya devolviendo los resultados a medida que los obtiene en vez de devolverlos todos juntos al final de su ejecución. Ése es el cometido de yield, el de retornar un valor de una secuencia de valores. Además, devuelve el "control" al código llamante, quien decidirá si seguir o no con la ejecución e, incluso, inyectar nuevos datos para modificar el proceso. Es el modo que tiene python de crear corrutinas, cuyo potencial se ha visto ampliado muchísimo con las últimas versiones de python y los procesos asíncronos. Es bastante complejo.

Pero volviendo a la pregunta, se puede establecer una analogía entre funciones y objetos. La definición de una función sería como tener una clase con un sólo método, y la ejecución de la función sería como crear instancias de la clase para crear un entorno de ejecución que llamamos "clausura" y que desaparece al finalizar la función. Al usar yield, interrumpimos la ejecución en ese punto, conservando la instancia para su uso posterior, así hasta que hayamos terminado.

Como que lo va a haciendo por partes conforme se vaya ocupando.

[ ]:
```
def contador(max):
    print("=Dentro de contador - empezando")
    n=0
    while n < max:
        print(f"=Dentro de contador - viene yield con n={n}")
        yield n
```

```
        print("=Dentro de contador - retomando después de yield")
        n=n+1
    print("=Dentro de contador - terminando")

print("Instanciando contador")
mycont = contador(3)
print("Contador instanciado")

for i in mycont:
    print(f"valor leido del iterador={i}")
print("Listo")
```

7 Closures

función la cual crea a su vez a otra de forma dinámica. Esta nueva función tiene memoria, siendo capaz de recordar, y utilizar, las variables lo locales y no locales definidas previamente. Funciones dentro de otra funcion anidadas

[31]:
```
def saludar_usuario(username):
    mensaje = 'Hola ' + username

    def saludar():
        print(mensaje)

    return saludar
funcion = saludar_usuario('Cody')
funcion()
```

```
Hola Cody
```

8 decoradores.

Un decorador en Python es una función que recibe otra función como parámetro, le añade cosas y retorna una función diferente.

Nos permiten envolver una función dentro de otra y modificar el comportamiento de esta última sin modificarla permanentemente. permite al usuario añadir nuevas funciones a un objeto existente sin modificar su estructura.

Los decoradores suelen aplicarse a las funciones, y desempeñan un papel crucial a la hora de mejorar o modificar el comportamiento de las funciones

Decorador no es más que una función la cual toma como input una función y asu vez retorna otra función.

Que queremos modificar el comportamiento de una función ya existente, pero sin tener que modificar su código. Esto es muy útil, principalmente, cuando queremos extender nuevas funcionalidades a dicha función. De allí el nombre decorar.

Como un pastel, donde, en ocasiones, la base del pastel (el pan) no es suficiente para una fiesta y debemos añadir elementos extras, quizás glaseado, velas, aderezos etc … de esta forma el pastel se verá mucho mejor y lo más importante sabrá mucho mejor.

Para decorar una función basta con colocar, en su parte superior de dicha función, el decorador con el prefijo @.

[22]:
```
def funcion_a(funcion_b):
    def funcion_c():
        print('Antes de la ejecución de la función a decorar')
        funcion_b()
        print('Después de la ejecución de la función a decorar')

    return funcion_c
@funcion_a
def saludar():
    print('Hola mundo!!')

print(saludar())
```

```
Antes de la ejecución de la función a decorar
Hola mundo!!
Después de la ejecución de la función a decorar
None
```

Ahora, ¿Qué pasa si nuestra función a decorar debe recibir argumentos y a su vez debe retornar algún valor? en estos casos haremos uso de los parámetros args y kwargs.

[20]:
```
def funcion_a(funcion_b):
    def funcion_c(*args, **kwargs):
        print('Antes de la ejecución de la función a decorar')
        result = funcion_b(*args, **kwargs)
        print(f'Después de la ejecución de la función a decorar')

        return result

    return funcion_c
@funcion_a
def suma(a, b):
    return a + b

Total = suma(20,30)
print(f'El total es de {Total}')
#print(suma(15, 15))
```

```
Antes de la ejecución de la función a decorar
Después de la ejecución de la función a decorar
El total es de 50
```

La función se ejecutará solo si dicha variable global es verdadera, y se dará un error de lo contrario.

Primero entra al decorador y luego entra a la base

[25]:
```
autenticado = True #True, False prueba con False y True una dice hola y la otra␣
 ↪El usuario no se ha autenticado

def requiere_autenticación(f):
    def funcion_decorada(*args, **kwargs):
        if not autenticado:
            print("Error. El usuario no se ha autenticado")
        else:
            return f(*args, **kwargs)
    return funcion_decorada

@requiere_autenticación
def di_hola():
    print("Hola Señor AUTENTICADO puede entrar al baile")

di_hola()
```

```
Hola Señor AUTENTICADO puede entrar al baile
```

9 3.3 Aplicación de Funciones en Problemas Reales

Deben investigar y desarrollar ejemplos de casos de uso reales en los que las funciones sean esenciales:

Un ejemplo en la vida real seria una funcion que calcule el impuestoa al valor agregado en Costa Rica

[11]:
```
def calcularIva(PrecioCompra,PorcentajeImpuesto = 0.13):
    IVA = PrecioCompra * PorcentajeImpuesto
    return IVA

PrecioDelaCompra = float(input('Digite el valor de la compra '))
Iva = calcularIva(PrecioDelaCompra,0.13)
Total = PrecioDelaCompra + Iva
print(f" El valor de la compra sin iva es de : {PrecioDelaCompra}")
print(f" El valor del iva es de : {Iva}")
print(f" El monto de la compra total  es de : {Total}")
```

```
El valor de la compra sin iva es de : 10000.0
El valor del iva es de : 1300.0
El monto de la compra total  es de : 11300.0
```

10 Aplicación en estructuras de datos (listas, diccionarios).

[ ]:
```python
# Crear una lista
frutas = ["manzana", "banana", "cereza"]

# Agregar un elemento a la lista
frutas.append("naranja")

# Acceder a un elemento por su índice
print(frutas[1])  # Output: banana

# Eliminar un elemento de la lista
frutas.remove("cereza")

# Recorrer la lista
for fruta in frutas:
    print(fruta)
```

Diccionarios Los diccionarios en Python son colecciones desordenadas de pares clave-valor. Son muy útiles para almacenar datos que se pueden asociar con una clave única. Aquí tienes un ejemplo de cómo trabajar con diccionarios:

[ ]:
```python
# Crear un diccionario
estudiantes = {
    "Juan": 25,
    "Ana": 22,
    "Luis": 23
}

# Agregar un par clave-valor
estudiantes["Maria"] = 24

# Acceder a un valor por su clave
print(estudiantes["Ana"])  # Output: 22

# Eliminar un par clave-valor
del estudiantes["Luis"]

# Recorrer el diccionario
for nombre, edad in estudiantes.items():
    print(f"{nombre} tiene {edad} años")
```

Combinando Listas y Diccionarios A menudo, es útil combinar listas y diccionarios para manejar datos más complejos. Aquí tienes un ejemplo de cómo hacerlo:

[ ]:
```python
# Lista de diccionarios
personas = [
    {"nombre": "Juan", "edad": 25, "ciudad": "San José"},
```

```
    {"nombre": "Ana", "edad": 22, "ciudad": "Heredia"},
    {"nombre": "Luis", "edad": 23, "ciudad": "Alajuela"}
]

# Agregar un nuevo diccionario a la lista
personas.append({"nombre": "Maria", "edad": 24, "ciudad": "Cartago"})

# Recorrer la lista de diccionarios
for persona in personas:
    print(f"{persona['nombre']} tiene {persona['edad']} años y vive en␣
 ↪{persona['ciudad']}")
```

11 Uso de funciones en procesamiento de datos.

El uso de funciones en Python es fundamental para el procesamiento de datos, ya que permite organizar el código, hacerlo más legible y reutilizable Supongamos que tienes un conjunto de datos de ventas y deseas calcular el total de ventas, el promedio y filtrar las ventas por encima de un cierto umbral Explicación: calcular_total(ventas): Esta función toma una lista de ventas y devuelve la suma total. calcular_promedio(ventas): Esta función calcula el promedio de las ventas. filtrar_ventas(ventas, umbral): Esta función filtra las ventas que son mayores que un valor umbral dado

[ ]:
```
# Datos de ejemplo
ventas = [150, 200, 300, 400, 250, 100, 350]

# Función para calcular el total de ventas
def calcular_total(ventas):
    return sum(ventas)

# Función para calcular el promedio de ventas
def calcular_promedio(ventas):
    return sum(ventas) / len(ventas)

# Función para filtrar ventas por encima de un umbral
def filtrar_ventas(ventas, umbral):
    return [venta for venta in ventas if venta > umbral]

# Uso de las funciones
total_ventas = calcular_total(ventas)
promedio_ventas = calcular_promedio(ventas)
ventas_filtradas = filtrar_ventas(ventas, 250)

print(f"Total de ventas: {total_ventas}")
print(f"Promedio de ventas: {promedio_ventas}")
print(f"Ventas por encima de 250: {ventas_filtradas}")
```

12 Optimización del rendimiento con funciones.

El rendimiento de las funciones en Python puede hacer que tu código sea más eficiente y rápido. Aquí tienes algunas estrategias y técnicas que puedes aplicar Las funciones incorporadas de Python están altamente optimizadas. Siempre que sea posible, utiliza funciones como sum(), min(), max(), etc., en lugar de escribir tus propias versiones

[30]:
```
total = 0
lista = [1, 2, 3]
for num in lista:
    total += num

# Usa esto:
total = sum(lista)
print(total)
```

6

[ ]:
```
# En lugar de esto:
for i in range(len(lista)):
    for j in range(len(lista)):
        if lista[i] == lista[j]:
            a = 0
            # hacer algo

# Usa esto:
elementos_vistos = set()
for elemento in lista:
    if elemento in elementos_vistos:
        a = 0 # hacer algo
    else:
        elementos_vistos.add(elemento)
```

13 Comparación entre funciones definidas por el usuario y funciones integradas (len(), sum(), etc.).

Si bien Python ya proporciona muchas funciones integradas como print() y len(), también puedes definir tus propias funciones para usar en tus proyectos.

Una de las grandes ventajas de usar funciones en tu código es que reduce el número total de líneas de código en tu proyecto. Es como personalizar funciones y ayudarnos con las funciones ya creadas.

14 Sección 3: Conclusiones

Ofreciendo una estructura clara, modularidad y eficiencia. Al dominar el manejo de funciones, los programadores pueden construir aplicaciones robustas, legibles y mantenibles. La práctica constante y la exploración de recursos de aprendizaje son claves para desarrollar habilidades sólidas en el manejo de funciones en Python.

15 Un resumen de hallazgos sobre la teoría y la práctica.

Me doy cuenta que en los libros tutoriales les falta abarcar temas y es ahi que tiene que buscar en mas lados para exclarecer dudas, muchas veces dan por sentado que se entendio un tema sin haberlo explicado.

16 Un análisis personal sobre qué aprendieron del uso de funciones en Python.

Es interesante es sencillo pero a la vez poco a poco se va complicando es cuestion de practicar, investigar, y dedicar muchas horas al aprendizaje hasta dominarlo por completo. Pienso que hay palabras reservadas que podrian ser mas intuitivas para hacer mas sencillo el aprendizaje.

17 Una sección de referencias con enlaces o libros consultados.

Libros de Python Python crash course third edition erick mattes.

Tutoriales en línea https://www.w3schools.com/python/python_functions.asp, https://www.programiz.com/python-programming/function, https://ellibrodepython.com/interes-compuesto