Status	Finished
Started	Friday, 22 November 2024, 11:18 AM
Completed	Friday, 22 November 2024, 11:23 AM
Duration	5 mins 40 secs
Marks	5.00/5.00
Grade	**10.00** out of 10.00 (**100**%)

Information

Funciones

Una función es un trozo de código organizado para realizar una tarea determinada y con un nombre que permite **llamarla** (invocar su ejecución). Una función se ejecuta solo cuando se la llama, y puede llamársela cuantas veces convenga, y desde múltiples puntos de un programa.

```
def greetings_func():
    """Muestra un saludo."""
    print("Hola")
```

El ejemplo anterior muestra la **definición** de una función en Python, que consta de una **cabecera** y de un **cuerpo**. La cabecera empieza con la palabra clave ***def*** seguida del **nombre** de la función. En el ejemplo, el nombre de la función es *greetings_func*. Los paréntesis que van después son necesarios. La cabecera de la función acaba con el carácter dos puntos.

El cuerpo de la función comienza en la línea siguiente a la cabecera y está desplazado un nivel de sangrado (un tabulador, o, preferiblemente, cuatro espacios) respecto a la misma. Estará formado por cualquier combinación de instrucciones, incluyendo sentencias de control, para realizar la tarea que deseemos.

Como primera línea del cuerpo de una función, justo debajo de la cabecera, es habitual poner un breve comentario en forma de una string encerrada entre **comillas triples**. Este comentario es una ***docstring***, una string de documentación, que se usa para proporcionar información básica sobre la función. Esta información puede ser usada por las aplicaciones de programación para proporcionar ayuda automática y autocompletado de código al usar la función. Se puede acceder a la *docstring* de una función usando el ***atributo*** ***__doc__*** tal como se muestra en el siguiente ejemplo:

```
print(greetings_func.__doc__)
```

El resultado del código anterior es:

```
Muestra un saludo
```

En general, los literales de string con comillas triples ocupan varias líneas:

```
    """Esto es un ejemplo de docstring multilínea.
       Las comillas de apertura y cierre
       deben estar alineadas.
    """
```

Para usar una función hay que invocarla (llamarla). La **llamada** a una función es como una instrucción más, que se forma poniendo el nombre de la función seguido de paréntesis:

```
def greetings_func():
    """Muestra un saludo"""
    print("Hola")
```

```
print("Antes de llamar a greetings_func")
greetings_func()  # Llamada a greetings_func
print("Después de llamar a greetings_func")
```

Al terminar la ejecución de una función, el flujo de ejecución **retorna** al punto justo después de la llamada, reanudándose la ejecución a partir de ahí. En cierto sentido, definir una función es como si hubiésemos añadido una nueva instrucción u operación al lenguaje, y la llamada sería su ejecución.

El resultado del código anterior es, pues:

```
Antes de llamar a greetings_func
Hola
Después de llamar a greetings_func
```

Question 1

Complete

Mark 1.00 out of 1.00

Elija la mejor definición del término "función"

Select one:

- Una tarea concreta que debe resolver un programa procesando unos datos para obtener un resultado
- Una secuencia de instrucciones adecuadamente organizadas para resolver una tarea determinada
- Un trozo de código organizado para realizar una tarea determinada y con un nombre que permite llamarla

Information

Funciones con parámetros

Las funciones pueden tener **parámetros**, que en Python funcionan como variables internas de la función durante su ejecución. Los parámetros de la función se inicializan con los datos pasados en la llamada. (datos de entrada necesarios para realizar la tarea). Por ejemplo:

```
def greetings_func(name):
    """Muestra un saludo.
       Parameter name: str - nombre de la persona a saludar
    """
    print("Hola", name)
```

La función `greetings_func` tiene ahora un parámetro, en este caso de nombre `name`, que se usa para incluir en el saludo el nombre de la persona a la que se saluda. Una función puede tener varios parámetros, cuyos nombres se escriben dentro de los paréntesis que siguen al nombre de la función, formando la denominada lista de **parámetros formales**. La lista de parámetros formales indica qué datos espera recibir la función cuando se la llame para realizar su tarea.

```
def consumo_medio(kilometros, litros):
    ...
```

Al llamar a una función que tiene parámetros, hay que pasarle **valores** para esos parámetros; estos valores se denominan **parámetros reales** (*actual parameters* en inglés, a veces llamados también *arguments* "argumentos"). El nombre de parámetros reales viene del hecho de que son los valores reales que se usan en la función en cada ocasión, frente a los formales, que son indicadores de los parámetros que se necesitan, pendientes de concretar en cada llamada:

```
name1 = "Pedro"
name2 = "Juan"
greetings_func(name1)
greetings_func(name2)
greetings_func("María" + " " + "Elena")
```

Como muestra el ejemplo, los parámetros reales pueden ser variables, valores literales, o, en general, cualquier expresión que dé como resultado un valor del tipo adecuado.

En el siguiente ejemplo, la función input devuelve una string (que ha tecleado el usuario), la cual a su vez la recibe greetings_func en su parámetro name:

```
greetings_func(input("Hola, ¿cómo te llamas? "))
```

Si la función tiene varios parámetros, hay que pasar valores para todos ellos; en Python, esto se puede hacer **por posición** (un parámetro real se asocia con el parámetro formal que ocupa su misma posición en la lista de parámetros) o **por nombre** (un parámetro real se asocia explícitamente con el nombre de un parámetro formal). El paso **por posición** es el habitual en la mayoría de los lenguajes de programación:

```
def consumo_medio(kilometros, litros):
    ...
```

```
consumo_medio(100, 6)  # Paso por posición -> (kilometros = 107, litros = 6)
consumo_medio(litros=17.5, kilometros=197)  # Paso por nombre
```

(Obsérvese que, tanto para los parámetros formales como para los actuales, se suele dejar un espacio después de la coma).

Los parámetros formales pueden tener **valores por omisión**, que se usan si en la llamada se omite dar un valor para ese parámetro. Los parámetros que no tienen valores por omisión no pueden omitirse en la llamada.

```
def consumo_medio(distancia, litros, unidades="kilometros"):
    ...
```

```
consumo_medio(103, 6.5)  # distancia = 103, litros= 6.5, unidades = "kilometros" (valor por omisión)
consumo_medio(87, 11, "millas")  # distancia = 87, litros= 11, unidades = "millas"
```

Los parámetros reales que estén después de uno o más que "se pasan" por omisión tienen que pasarse obligatoriamente por nombre, ya que se pierde la correspondencia entre la lista de parámetros formales y la lista de parámetros reales.

Es importante comprender que la relación en Python entre parámetros reales y formales consiste en que de los reales se toman sus valores, los cuales se asignan al comienzo de la ejecución de la llamada a los parámetros formales, que son variables internas de la función, que desaparecen al retornar la ejecución de esta. El que en una llamada usemos variables para algunos parámetros reales es irrelevante (lo que nos importa es el valor que contienen en el momento de la llamada) en el sentido de que estas variables no tienen relación con los parámetros formales, incluso si les diésemos el mismo nombre, p.e. en una llamada f(x), x será una variable en el contexto en que se genera la llamada, y aunque al parámetro formal de f podríamos haberlo llamado también x "por coincidencia", no existe confusión ni conflicto, pues este parámetro formal x es una variable interna a la función distinta de la otra variable también llamada x existente en el punto de la llamada, de forma que incluso si durante la ejecución de la función le asignáramos un nuevo valor al parámetro formal x, la otra variable x no se vería afectada.

Question 2

Complete

Mark 1.00 out of 1.00

```
def my_func(a, b, c):
    k = a
    l = b  + c
    m = a + b + c
...
x = 1
y = 2
z = 3
my_func(x, y, z)
```

¿Cuáles son los parámetros formales de la función *my_func*?

Select one:

- k, l, m
- a, b, c
- x, y, z

Information

Funciones que devuelven un resultado

Además de poder admitir datos como parámetros, las funciones pueden devolver un resultado al finalizar su tarea. Esto es coherente (además del nombre) con que las funciones resuelven problemas del tipo hallar un dato desconocido a partir de otros datos conocidos, aplicando un **algoritmo**, que describe la solución de un problema en función de los datos necesarios para representar un caso concreto del problema y de los pasos necesarios para obtener el resultado deseado.

Para devolver un resultado se utiliza la instrucción ***return*** con el valor a devolver (el cual se puede expresar en forma de un literal del tipo apropiado, una variable que lo contenga o, en general, cualquier expresión, cuya evaluación nos dará el valor a devolver):

```
def suma(a, b):
    """Suma los valores de sus parámetros."""
    result = a + b
    return result
```

El resultado devuelto por una función se puede usar directamente en una expresión (es como si lo hubiésemos escrito en lugar de la llamada, aunque con la flexibilidad de que dicho valor será uno u otro, p.e. según los valores de los parámetros en ese momento):

```
print(suma(a, b))
...
if 10 * suma(a, b) > 50:
    ...
```

También se puede asignar a una variable para usarlo más tarde:

```
x = suma(a, b)
```

(Recuérdese que, aunque hayamos empleado iguales nombres para las variables usadas como parámetros reales y para los parámetros formales de *suma*, ello no supone que haya relación alguna entre ellas pues, aunque compartan nombre, serán variables distintas).

Dado que al ejecutarse la instrucción *return* la función finaliza su tarea, cualquier instrucción que se encuentre después del *return,* en su mismo nivel de anidamiento, no se ejecutará nunca. Es decir, no deben escribirse instrucciones detrás de un *return*:

```
def suma(a, b):
    """Suma los valores de sus parámetros."""
    result = a + b
    return result
    result = 0
    print(result)
```

```
def PositivoNegativo(num):
    """Devuelve si el número es positivo, negativo o cero."""
    if num < 0:
        return "negativo"
        num = 0
        print(num)
    elif num > 0:
        return "positivo"
        num = 0
        print(num)
    return "cero"
    num = 0
    print(num)
```

En realidad, en Python todas las funciones devuelven un valor, solo que las que no lo hacen explícitamente mediante *return* devuelven el valor ***None***, que es el único valor del tipo de datos *NoneType* y se interpreta como "Ningún valor". Por ejemplo, el siguiente código:

```
def greetings_func():
    """Muestra un saludo."""
    print("Hola")

result = greetings_func();
print("Resultado devuelto por la función:", result);
```

produce el siguiente resultado:

```
Hola
Resultado devuelto por la función: None
```

La palabra "Hola" la escribe la función, que devuelve *None*. El valor *None* devuelto por la función es asignado a la variable *result* y mostrado por la instrucción *print* que se ejecuta en la siguiente línea.

La instrucción *return* puede utilizarse como cualquier otra instrucción dentro de una función, esto es, puede haber varios return en distintos puntos de la misma, eso sí, una vez que se alcanza un return, termina la ejecución de la función (por lo que no tiene sentido escribir una instrucción del mismo nivel de sangrado inmediatamente después).

Question 3

Complete

Mark 1.00 out of 1.00

```
def my_func(a, b, c):
    if a > b:
        return c
    elif a > c:
        return b
    else:
        return a

x = 13
y = 3
z = 23
r = my_func(x, y, z)
```

¿Qué valor se asigna a la variable r?

Select one:

- 13
- 23
- 3

Question 4

Complete

Mark 1.00 out of 1.00

```
def my_func(a, b, c):
    if a > b:
        return c
    elif a > c:
        return b
    else:
        return a

x = 13
y = 3
z = 23
r = my_func(c = x, a = y, b = z)
```

¿Qué valor se asigna a la variable r?

Select one:

- 13
- 3
- 23

Information

Funciones polimórficas

Muchos lenguajes de programación exigen que se declaren los tipos de los datos que pueden asociarse a una variable o pasarse a un parámetro formal o que devuelva una función. Python tiene **tipado dinámico**, lo que significa que no requiere declaraciones de esta clase: el tipo de dato asociado a una variable lo determina el del valor que se e asigna en el momento de crearla, y un parámetro formal puede asociarse a un tipo de dato diferente en cada llamada. Esto, en el caso de las funciones, permite que sean **polimórficas** por definición (una función polimórfica es aquella que se puede usar con distintos tipos de parámetros y/o distinto número de parámetros). Una función en Python tal como:

```
def sum(a, b):
    """Suma los valores de sus parámetros"""
    result = a + b
    return result
```

puede aplicarse sin problema a cualquier pareja de valores que admitan el uso del operador +, entre ellos:

```
print(sum(1, 3))              # Números enteros
print(sum(3.5, 4.8))          # Números reales
print(sum(3.5, 4))            # Un número entero con un real
print(sum("Hola ", "mundo"))  # Strings, que serán concatenadas con el +
print(sum(3+2j, 5-1j))        # Números complejos
```

Resultado de la ejecución anterior:

```
4
8.3
7.5
Hola mundo
(8+1j)
```

Information

Anotación de los tipos de los parámetros de una función

Aunque el polimorfismo proporcionado a Python por el tipado dinámico puede ser una potente herramienta (permite que una función se use en situaciones muy diferentes, sin tener que reescribir múltiples versiones), cuando no es nuestra intención escribir una función polimórfica puede ser útil explicitar qué tipo de dato se supone que debe recibir un parámetro formal, o qué tipo de resultado se supone que va a devolver una función:

```
def sum(a: int, b: int) -> int:
    """Suma los valores de sus parámetros."""
    result = a + b
    return result
```

Como se muestra en el ejemplo anterior, para los tipos más básicos (int, float, bool, str) basta con poner dos puntos detrás del parámetro y añadir el tipo esperado. Para otros tipos de datos, que se irán viendo en otras lecciones, puede ser necesario incluir alguna instrucción adicional. El tipo del resultado de la función se explicita después de una "flecha" añadida al final de la cabecera, antes de los dos puntos.

El primer efecto del tipado explícito es informativo. Normalmente, en la *docstring* de una función se pone información sobre sus parámetros, incluyendo de qué tipo se espera que sean, como en el siguiente ejemplo:

```
"""Suma los valores de sus parámetros.
   Parámetros:
   a: (int) - Primer sumando
   b: (int) - Segundo sumando
"""
```

Si se explicitan los tipos en la lista de parámetros formales y, además, se usan nombres adecuados, la *docstring* puede reducirse sustancialmente, ya que mucha información relevante está incluida en la propia cabecera de la función.

```
def sum(primer_sumando: int, segundo_sumando: int) -> int:
    """Suma los valores de sus parámetros"""
    result = primer_sumando + segundo_sumando
    return result
```

Además de esto, se pueden usar herramientas específicas que, basándose en la información proporcionada por el tipado explícito, son capaces de detectar por anticipado posibles errores que, de otro modo, sólo se manifestarían al ejecutar el programa.

Question 5

Complete

Mark 1.00 out of 1.00

```
def my_func(a: int, b: float) -> bool:
    ...
```

¿De qué tipo es el valor que se espera devuelva la función *my_func*?

Select one:

- bool
- int
- float