6. Funciones

6.1. Declaración

Para declarar una función en Python se ha uso de la palabra de reservada `def` seguida del nombre y los argumentos entre paréntesis, al igual que con otras estructuras los dos puntos indican que las instrucciones siguientes que estén indentadas hacen parte de la función, por último, se emplea la palabra `return` para indicar la salida de esta.

```
def nombre_funcion(argumento_1, ..., argumento_n):
    indicación_1
    indicación_2
    ...
    indicación_n

    return salida
```

6.1.1. Tipos de Declaración

Las funciones pueden ser clasificadas según la existencia de sus entradas y salidas, debido a que tanto los argumentos como la el retorno son opcionales. A continuación, se mostrarán los cuatro tipos de declaraciones.

- **Función completa:** Es la que comparte un mayor parecido al concepto matemático de función, debido a que tiene unas entradas (argumentos), las transforma (con las indicaciones) y devuelve un resultado (salida), normalmente este resultado luego será usado en el código por lo cual se guarda en una variable. Este tipo de declaración se usa por ejemplo en las funciones de lectura y almacenamiento de datos y en las funciones de los módulos.

```
# Declaración

def tri_cua_per(x,y):
    "Devuelve el trinomio cuadrado perfecto, de los números indicados"
    resultado=(x**2)+(2*x*y)+(y**2)

    return resultado

# Uso
num_1=int(input("Por favor ingrese un número: "))
num_2=int(input("Por favor ingrese otro número: "))

trinomio=tri_cua_per(num_1,num_2)

print("El trinomio cuadrado perfecto de",num_1,"y",num_2,"es: ",trinomio)
```

```
# Pase el código anterior aquí.
```

- **Función sin salidas:** Esta función toma los argumentos de entrada y realiza una transformación de estas pero no devuelve ningún objeto, normalmente es empleada para mostrar objetos o guardar archivos.

```
# Declaración

def tri_cua_per(x,y):
    "Devuelve un mensaje con el trinomio cuadrado perfecto, de los números indicados"

    resultado=(x**2)+(2*x*y)+(y**2)
    print("El trinomio cuadrado perfecto de",x,"y",y,"es: ",resultado)
```

```
    return None # Si esta línea no está escrita el intérprete de Python hará de cuenta que si l

# Uso
num_1=int(input("Por favor ingrese un número: "))
num_2=int(input("Por favor ingrese otro número: "))

tri_cua_per(num_1,num_2)
```

```
# Pase el código anterior aquí.
```

- **Función sin entradas:** Este tipo de función en lugar tomar sus entrada con los argumentos utiliza variables globales (variables que ya existen en código donde se va a usar la función), o puede pedir directamente las entradas al usuario con `input`, al igual que una función completa su salida puede ser guardada en una variable aparte para ser utilizada posteriormente. Algunas funciones de validación usan esta forma de declaración.

La función usando variables globales sería así:

```
# Declaración

def tri_cua_per():
    "Devuelve el trinomio cuadrado perfecto de las variables num_1 y num_2"

    resultado=(num_1+num_2)**2

    return resultado

# Uso

num_1=int(input("Por favor ingrese un número: "))
num_2=int(input("Por favor ingrese otro número: "))
trinomio=tri_cua_per()

print("El trinomio cuadrado perfecto de",num_1,"y",num_2,"es: ",trinomio)
```

```
# Pase el código anterior aquí.
```

La función pidiendo directamente las entradas seria así:

```
# Declaración

def tri_cua_per():
    "Pide dos números y los devuelve junto con su trinomio cuadrado perfecto"

    x=int(input("Por favor ingrese un número: "))
    y=int(input("Por favor ingrese otro número: "))
    resultado=(x+y)**2

    return (x,y,resultado) # Se devuleve una tupla

# Uso

num_1,num_2,trinomio=tri_cua_per() # Aquí se abre la tupla asignando una variable a cada elemer
print("El trinomio cuadrado perfecto de",num_1,"y",num_2,"es: ",trinomio)
```

```
# Pase el código anterior aquí.
```

- En el ejemplo anterior las variables 'x' y 'y' tienen que devolverse como salida y almacenarse en las variables num_1 y num_2 porque son variables locales de la función, por lo cual, no existen en otra parte del código y son utilizadas fuera de la función al correr el código mostrará NameError.

```
# Mini_Ejemplo: Variables locales

def tri_cua_per():
    "Pide dos números y los devuelve junto con su trinomio cuadrado perfecto"

    x=int(input("Por favor ingrese un número: "))
    y=int(input("Por favor ingrese otro número: "))
    resultado=(x+y)**2

    return (resultado) # Se devuelve una tupla

# Uso

trinomio=tri_cua_per()
print("El trinomio cuadrado perfecto de",x,"y",y,"es: ",trinomio)

>>> "NameError: name 'x' is not defined"
```

- **Función sin entradas ni salidas:** Esta tipo de declaración es una mezcla de las dos anteriores, ya que, sus entradas la pide directamente al usuario o usa variables globales y sus salidas solo

pueden mostradas o guardadas en un archivo. La función de este tipo más conocida es la función del programa principal main( ).

```
# Declaración

def tri_cua_per():
    "Pide dos números y devuelve un mensaje con su trinomio cuadrado perfecto"

    x=int(input("Por favor ingrese un número: "))
    y=int(input("Por favor ingrese otro número: "))

    resultado=(x+y)**2
    print("El trinomio cuadrado perfecto de",x,"y",y,"es: ",resultado)


    return None # Si esta línea no está escrita el intérprete de Python hará de cuenta que si l

# Uso

tri_cua_per()
```

```
# Pase el código anterior aquí.
```

6.1.2. Función Lambda

La función lambda permite definir pequeñas funciones de una sola línea directamente en el código que va a hacer uso de ella.

Toda función lambda se puede expresar como una función convencional, pero no toda función convencional se puede expresar como una función lambda.

La sintaxis para crear una función de este tipo es:

```
Nombre= lambda <parámetros>:<indicación>
```

La función tri_cua_per completa vista en apartado anterior se puede escribir como función lambda.

```
# Mini_Ejemplo: Función Lambda

# Declaración
```

```
tri_cua_per = lambda x,y: (x+y)**2

# Uso

num_1=int(input("Por favor ingrese un número: "))
num_2=int(input("Por favor ingrese otro número: "))

trinomio=tri_cua_per(num_1,num_2)

print("El trinomio cuadrado perfecto de",num_1,"y",num_2,"es: ",trinomio)
```

```
# Pase aquí el mini ejemplo anterior
```

6.1.3. Documentación

Documentar una función es importante para facilitar la lectura de código y al momento de crear y utilizar funciones de módulos propios. Para hacerlo se debe escribir en la primera línea de la función una descripción de lo que hace, las entradas que requiere y las salidas regresa, haciendo uso de las comillas simples, dobles o triples, dependiendo de qué tan grande sea la descripción.

Como se vio en capítulos anteriores para ver la documentación de un objeto se usa la función `help`.

```
def dif_cua (x,y):
    "Devuelve la diferencia de cuadrados de los números indicados"
    resultado=(x**2)-(y**2)

    return resultado

help(dif_cua)
```

```
# Pase el código anterior aquí.
```

6.2. Argumentos

Una función puede tener dos tipos de argumentos: obligatorios y opcionales. A continuación, se mostrarán las características de cada uno.

6.2.1. Argumentos Obligatorios

Como su nombre lo dice son argumentos que deben ser ingresados obligatoriamente en la función para que esta pueda ejecutarse, de lo contrario aparecerá un error. La característica más importante es que deben ser inicializados en el mismo orden en que fueron declarados, de lo contrario, se pueden tener resultados indeseados o errores al correr el código.

```
# Declaración

def resta(n1,n2):
    "Devuelve la resta de n1-n2"

    return n1-n2

print("Inicialización correcta\n")
print("923 - 45 es:",resta(923,45))

print("\nInicialización incorrecta\n")
print("923 - 45 es:",resta(45,923))
```

```
# Pase el código anterior aquí.
```

```
# Declaración

def peso(nombre,p_real,p_meta):
    "Devuelve un mensaje al usuario diciéndole cuanto tiene que bajar o subir para llegar a su

    brecha=p_real-p_meta

    if brecha<0:
        print(nombre+": Debe subir",brecha,"Kg para llegar a los", p_meta,"Kg")
    elif brecha==0:
        print("Felicitaciones ha cumplido su meta de peso")
    else:
        print(nombre+": Debe bajar",brecha,"Kg para llegar a los", p_meta,"Kg")

    return None

print("Inicialización correcta\n")
peso("Julian",95,80)
```

```
# Pase el código anterior aquí.
```

```
# Mini_Ejemplo: Orden Incorrecto de Argumentos

# Declaración

def peso(nombre,p_real,p_meta):
    "Devuelve un mensaje al usuario diciéndole cuanto tiene que bajar o subir para llegar a su

    brecha=p_real-p_meta

    if brecha<0:
        print(nombre+": Debe subir",brecha,"Kg para llegar a los", p_meta,"Kg")
    elif brecha==0:
        print("Felicitaciones ha cumplido su meta de peso")
    else:
        print(nombre+": Debe bajar",brecha,"Kg para llegar a los", p_meta,"Kg")

    return None

print("\nInicialización incorrecta\n")
peso(45,"Alicia",50)

>>> "TypeError: unsupported operand type(s) for -: 'str' and 'int'"
```

6.2.2. Argumentos Opcionales

Son argumentos que pueden ser inicializados o no al usar la función, porque se les asignó un valor predeterminado al momento de declarar la función y si no son indicados la función operará con estos valores.

Si la función tiene los dos tipos de argumentos, los obligatorios siempre deben ir primero que los opcionales, además estos últimos, aunque tienen la característica de que se pueden inicializar según su posición también se puede hacer por asignación directa con su nombre lo que permite una inicialización diferente al orden preestablecido.

En el siguiente Mini_Ejemplo los argumento menor, n y menor son opcionales.

```
# Mini_Ejemplo: Argumentos Opcionales 1

import random
```

```
def l_aleatorios(mayor,menor=1,n=10,nd=2):
    """
    Devuelve una lista de n números aleatorios entre el rango indicado,
    se debe ingresar obligatoriamente número mayor del rango, por defecto el número menor es 1,
    la cantidad de números es 10 y el número de decimales es 2.
    """

    d=10**nd
    aleatorios=[]
    for i in range(n):
        aleatorios.append(random.randint(menor*d,mayor*d)/d)

    return aleatorios

lista_aleatoria=l_aleatorios(100)
print("La lista de números aleatorios es: ",lista_aleatoria)
```

```
# Pase aquí el mini ejemplo anterior
```

```
# Mini_Ejemplo: Argumentos Opcionales 2

import random

def l_aleatorios(mayor,menor=1,n=10,nd=2):
    """
    Devuelve una lista de n números aleatorios entre el rango indicado,
    se debe ingresar obligatoriamente número mayor del rango, por defecto el número menor es 1,
    la cantidad de números es 10 y el número de decimales es 2.
    """

    d=10**nd
    aleatorios=[]
    for i in range(n):
        aleatorios.append(random.randint(menor*d,mayor*d)/d)

    return aleatorios

lista_aleatoria=l_aleatorios(4000,900,nd=4,n=15)
print("La lista de números aleatorios es: ",lista_aleatoria)
```

```
# Pase aquí el mini ejemplo anterior
```

6.3. Funciones Recursivas

Las funciones recursivas son las que se utilizan a si mismas, funcionando como un estructura de repetición, sin embargo, se debe poner especial atención en el diseño para que deje de ejecutarse cuando sea necesario, de lo contrario, el programa podría entrar en bucle infinito como sucede con el ciclo while.

A continuación se muestra una función recursiva para calcular el n_avo Fibonacci solicitado por el usuario, como se vio en el Mini_Ejemplo de la Sucesión de Fibonacci del capítulo 4.

```
# Mini_Ejemplo: Función Recursiva

# Declaración

def s_fibonacci (sucesion=[],n=100):
    if len(sucesion)==0:
        sucesion.extend([[0,0],[1,1]])
        print("El fibonacci No.0 es 0\n\nEl fibonacci No.1 es 1\n")

    if (len(sucesion)<=n):
        fibonacci=sucesion[-2][1]+sucesion[-1][1]
        sucesion.append([sucesion[-1][0]+1,fibonacci])
        print("El fibonacci No."+str(sucesion[-1][0])+" es", fibonacci,"\n")
        s_fibonacci(sucesion,n)

#Usar
n_avo=input("Por favor ingrese el n-avo Fibonacci hasta donde quiere calcular la sucesión, debe

if n_avo.isnumeric() and n_avo!="0":
    s_fibonacci(n=int(n_avo))
else:
    print("No ha ingresado un número válido, por defecto se mostrará hasta el Fibonacci No. 100
    s_fibonacci()
```

```
# Pase aquí el mini ejemplo anterior
```

6.4. Validaciones

Las validaciones son funciones que ayudan a que el programa detecte errores potenciales en el momento en el que usuario ingrese los datos, asegurándose que el ingreso sea correcto para el desarrollo normal del programa.

Antes de ver como es la estructura general de una función de validación se deben conocer las excepciones, las cuales se explicarán a continuación.

6.4.1. Excepciones

Las excepciones son una herramienta de Python para detectar algunos errores, por ejemplo, cuando un bloque de código detecta un error lanza una excepción asociada a este y espera a que otro bloque la capture y la maneje, si este último no existe el programa deja de ejecutarse y muestra un mensaje de error con la excepción lanzada.

Hay excepciones que tienen que ver con la forma de escribir código, como **SintaxisError** o **IndentationError**, sin embargo, estas tienen que ver más con el lenguaje de programación y errores del desarrollador, por lo cual no se van a tratar demasiado a fondo en esta guía.

A continuación, hay una lista de las excepciones más comunes con las que se puede encontrar desarrollando esta guía, una breve descripción de cuando son lanzadas y algunos ejemplos.

- **AssertionError:** Es lanzado cuando se utiliza assert y la expresión lógica que evalúa arroja False, el uso de assert se verá más adelante en el apartado de pruebas unitarias.

```
assert 1=="Gatos" | assert "4.5".isnumeric() | assert 10 in (9,8,3,2)
```

- **AttributeError:** Se presenta cuando se usa un método que no le pertenece a un objeto.

```
(4,7,8.1,"Perros").append("Laura") | ["Luis","Lorena","Lola","Lucas"].isalpha() | {"Color":"Neg
```

- **IndexError:** Como se vio en capítulo 4, este error se da porque se utiliza un índice que no existe en un iterable, es decir un índice mayor o igual a tamaño de este.

```
tupla= (4,7,8.1,"Perros")

tupla[len(tupla)] | tupla[11] | tupla[-6] |
```

- **ImportError:** Ocurre cuando se importa un módulo que no existe o que no se ha instalado.

```
import NUMPY | import stadistics | import plt |
```

- **KeyError:** Se muestra cuando se intenta acceder a un elemento de un diccionario con una calve que no existe.

```
diccionario={"Color":"Negro","Opacidad":0.3}

diccionario["opacidad"] | diccionario["cantidad"] | diccionario["relleno"]
```

- **NameError:** Aparece cuando se utiliza un objeto que no existe o variable local fuera de su bloque como se vio en el Mini_Ejemplo Variables Locales del apartado 6.1.1.

```
c=100+a | len(l) | f_fantasma(98) | diccionario={"Color":Negro,"Opacidad":0.3}
```

- **TypeError:** Es mostrado cuando el tipo de objeto empelado en una función o en una operación no es compatible con esta, como se mostró en el Mini_Ejemplo del apartado anterior 6.2.1.

```
len(9) | "Animales"/2 | [2,3]-[4,3] | range(4.5) | float(3,4) | sorted([1,3,9,"cadena","c"])
```

- **VlaueError:** Es mostrado cuando el valor del objeto usado en una función o en una operación no es válido a pesar de que su tipo si lo sea.

```
int("2.9") | float("3,4") | float("cadena") |
```

- **ZeroDivisionError:** Se genera cuando se divide un número entre 0.

```
67/0 | 0.000438/0 |
```

6.4.2. Validación de Entradas

La estructura básica de una función de validación de entradas es:

```
def validacion ():
    # Intentar ejecutar el bloque que puede generar un error
    try:
        Instrucciones que pueden generar un error
        ...
        return salidas

    # Verificar si ha lanzado una excepción
```

```
    except (excepción_1, excepción_2):
        Instrucciones si se lanzó una excepción
        ...
        validacion()
```

Si no se tiene claro que excepción puede arrojar el bloque, entonces se utiliza la palabra reservada `exception` para cubrir todas las excepciones estándar de Python.

A continuación, se muestran algunos ejemplos de validación de entradas.

- **Validación Números Enteros**

```
# Min_Ejemplo: Validación Números Enteros

def v_num_entero(mensaje):
    "Pide un número entero mostrando el mensaje indicado, hasta que el valor ingresado sea váli

    try:
        num=int(input(mensaje))
        print("El valor ingresado es correcto, muchas gracias.")
        return num
    except(ValueError):
        print("""
        \n---------------------------------------------------------------------\n
        Ha ingresado un valor inválido vuelva a intentarlo.
        \n---------------------------------------------------------------------\n
        """)
        return v_num_entero(mensaje)

m="Por favor ingrese su edad en años: "

edad=v_num_entero(m)
```

```
# Pase aquí el mini ejemplo anterior
```

- **Validación Números Decimales**

```
# Min_Ejemplo: Validación Números Decimales

def v_num_decimal(mensaje):
    "Pide un número decimal mostrando el mensaje indicado, hasta que el valor ingresado sea vál

    try:
```

```
        num=input(mensaje)
        num=float(num)
        print("\nEl valor ingresado es correcto, muchas gracias.")
        return num
    except(ValueError):
        print("""
        \n---------------------------------------------------------------------\n
        Ha ingresado un valor inválido vuelva a intentarlo.
        \n---------------------------------------------------------------------
        """)

        indice=num.index(",")

        if num[:indice].isnumeric() and num[indice+1:].isnumeric():
            print("""
        Recuerde que el decimal se índica con punto ( . ).
        \n---------------------------------------------------------------------\n
        """)

        return v_num_decimal(mensaje)

m="Por favor ingrese su estatura en metros, por ejemplo, 1.70: "

estatura=v_num_decimal(m)
```

```
# Pase aquí el mini ejemplo anterior
```

- **Validación de Rangos**

Como se ve en siguiente ejemplo, para lanzar una excepción del sistema cuando cumplen ciertas condiciones se usa la palabra reservada `raise`, indicando el nombre de la excepción y si se requiere entre paréntesis se puede poner un mensaje, indicando el error, para que se muestre en caso de que la excepción no sea capturada por un bloque de código.

```
# Min_Ejemplo: Validación de Rangos

def v_rango(mensaje,minimo,maximo):
    "Pide un número dentro de un rango según el mensaje indicado, hasta que el valor ingresado

    lineas=("\n---------------------------------------------------------------------\n")
    try:
        num=input(mensaje)
        num=float(num)
        try:
```

```
            if not minimo<=num<=maximo:
                raise ValueError()
            print("\nEl valor ingresado es correcto, muchas gracias.")

        except(ValueError):
            print(lineas+"Recuerde ingresar un valor dentro del rango indicado."+lineas)

            return v_rango(mensaje,minimo,maximo)

    except(ValueError):
        print(lineas+"No ingresado un valor inválido vuelva a intentarlo"+lineas)

        return v_rango(mensjae)

minimo= 100
maximo= 190
m="Por favor ingrese un número entre "+str(minimo)+" y "+str(maximo)+": "

valor=v_rango(m, minimo, maximo)
```

```
# Pase aquí el mini ejemplo anterior
```

Anterior - *Siguiente*

Home