**Módulo 9:**

**Funciones propias**:

* + def return
  + Argumentos posicionales y por omisión
  + Parámetros arbitrarios en Python
  + Funciones con parámetros y operadores
  + Orden de los argumentos
  + Recursividad
  + Anotaciones en funciones
  + docstrings
  + Retorno Empaquetados y desempaquetados.
  + Ámbitos de los distintos objetos.

|  |
| --- |
| Por flujo de información las funciones deben estar antes de ser llamadas.  Son objetos y deben cumplir con las normas de nombres de Python interprete y comunidad.  Verificar el ámbito de:  Variables para lectura y escritura en la función  Colecciones para lectura y escritura en la función  Empaquetado de entrada y salida |

Función es un objeto con nombre propio con un bloque de código reutilizable.

Este bloque se utiliza para realizar una tarea específica y permiten dividir el código en fragmentos más pequeños y lógicos,

Estos bloques o funciones permiten la organización, la reutilización y el mantenimiento del código.

Se puede asignar a un encargado (persona grupo) a cada función.

Para crear una función se utiliza la palabra clave **def**, seguida del nombre de la función (en snake case, sin usar palabras claves de Python),

Luego del nombre paréntesis que pueden contener (o no) los objetos parámetros para recibir datos desde afuera de la función a dentro de esta.

Cuando se llama a una función, se utiliza el nombre de la función seguido de paréntesis. Los argumentos pasados en la llamada a la función se asignan a los parámetros definidos en la declaración de la función.

En Python no es totalmente cierto que se debe proveer el mismo número de argumentos como haya parámetros definidos. Veremos mas adelante parámetros con argumentos por omisión (donde algunos parámetros no se envían a la función, se omiten)

Se utiliza la función **return** para devolver un objeto al lugar que lo llamo.

|  |
| --- |
| **Programación funcional**  teóricos |
| Las funciones en Python son bloques de código reutilizables que realizan tareas específicas. Tienen varias características que las hacen poderosas y flexibles en la programación. A continuación se presentan algunas de las características clave de las funciones en Python:   * Modularidad: Las funciones permiten dividir el código en módulos más pequeños y manejables. Esto mejora la legibilidad, el mantenimiento y la reutilización del código, ya que una función puede ser llamada en múltiples lugares dentro de un programa. * Reutilización de código: Las funciones permiten escribir una vez y utilizar muchas veces. Puedes definir una función y llamarla en diferentes partes de tu programa sin tener que volver a escribir el mismo código una y otra vez. * Encapsulación: Las funciones encapsulan la lógica y las operaciones en un solo lugar. Puedes agrupar un conjunto de instrucciones relacionadas en una función, lo que facilita el seguimiento y la comprensión del flujo del programa. * Parámetros: Las funciones pueden aceptar parámetros, que son valores que se pasan a la función para que los utilice en sus operaciones. Los parámetros permiten que una función sea más genérica y flexible, ya que pueden tomar diferentes valores en cada llamada. * Retorno de valores: Las funciones pueden devolver un valor o resultado después de realizar sus operaciones. Esto permite que el resultado de una función se utilice en otras partes del programa o se asigne a una variable para su posterior procesamiento. * Ámbito (scope) de las variables: Las variables definidas dentro de una función tienen un ámbito local, lo que significa que solo son accesibles dentro de la función en la que se definen. Esto evita conflictos de nombres con variables fuera de la función y ayuda a mantener la claridad y la integridad del código. * Funciones como objetos de primera clase: En Python, las funciones son objetos de primera clase, lo que significa que se pueden asignar a variables, pasar como argumentos a otras funciones y devolver como resultados de otras funciones. Esto permite un alto grado de flexibilidad en la programación y la implementación de conceptos como funciones de orden superior y programación funcional.   Estas son algunas de las características más importantes de las funciones en Python. Con el uso adecuado de las funciones, puedes modularizar tu código, mejorar su mantenibilidad y reutilización, y escribir programas más estructurados y legibles. |

La sintaxis básica de una función en Python es la siguiente:

**def** mi\_funcion (parametro1, parametro2, ...):

# Código de la función

**return** # Opcionalmente, la función puede retornar un valor

mi\_funcion (parametro1, parametro2, ...)

·

|  |
| --- |
| def saludar(nombre):  mensaje = f"Hola, {nombre}!"  print(mensaje)  saludar("Pepe") |

Salida esperada por consola

|  |
| --- |
| Hola, Pepe! |

·

El argumento “Pepe” se envia a la función saludar donde ingresa mediante el objeto nombre se concatena en mensaje y lo imprime.

El orden de los argumentos posicionales es importante.

Además de incluir el número correcto de argumentos, es importante notar el orden en el cual se indican los argumentos.

Los argumentos necesitan ser escritos en el orden exacto, en el que se han declarado los parámetros en la definición de la función.

Los parámetros sólo existen dentro de las funciones en donde han sido definidos, y el único lugar donde un parámetro puede ser definido es entre los paréntesis después del nombre de la función, donde se encuentra la palabra reservada def.

La asignación de un valor a un parámetro de una función se hace en el momento en que la función es invocada, especificando el argumento

·

|  |
| --- |
| def sumar(dato\_1,dato\_2):  salida= dato\_1+dato\_2  return salida  #---------------------------------------------------------------------------------------------  dato\_primero=9  dato\_segundo=2  entrada= sumar(dato\_primero,dato\_segundo)  print (f"sumar {dato\_primero} + {dato\_segundo} = {entrada}") |

Salida esperada por consola

|  |
| --- |
| sumar 9 + 2 = 11 |

En este caso, la función sumar() toma dos parámetros dato₁ Y dato\_2, realiza la suma y retorna el resultado y se imprime en la pantalla.

·

|  |
| --- |
| def sumar(dato\_1,dato\_2):  salida= dato\_1+dato\_2  return salida  def restar(dato\_1,dato\_2):  salida= dato\_1-dato\_2  return salida    def dividir(dato\_1,dato\_2):  salida= dato\_1/dato\_2  return salida    def multiplicar(dato\_1,dato\_2):  salida= dato\_1\*dato\_2  return salida  def potenciar(dato\_1,dato\_2):  salida= dato\_1\*\*dato\_2  return salida    def radicar(dato\_1,dato\_2):  salida= dato\_1\*\*(1/dato\_2)  return salida  #---------------------------------------------------------------------------------------------  dato\_primero=9  dato\_segundo=2  entrada= sumar(dato\_primero,dato\_segundo)  print (f"sumar {dato\_primero} + {dato\_segundo} = {entrada}")  entrada= restar(dato\_primero,dato\_segundo)  print (f"restar {dato\_primero} - {dato\_segundo} = {entrada}")    entrada= dividir(dato\_primero,dato\_segundo)  print (f"dividir {dato\_primero} / {dato\_segundo} = {entrada}")    entrada= multiplicar(dato\_primero,dato\_segundo)  print (f"multiplicar {dato\_primero} \* {dato\_segundo} = {entrada}")  entrada= potenciar(dato\_primero,dato\_segundo)  print (f"potenciar {dato\_primero} \*\* {dato\_segundo} = {entrada}")    entrada= radicar(dato\_primero,dato\_segundo)  print (f"radicar {dato\_primero} \*\* (1/{dato\_segundo}) = {entrada}") |

Salida esperada por consola

|  |
| --- |
| sumar 9 + 2 = 11  restar 9 - 2 = 7  dividir 9 / 2 = 4.5  multiplicar 9 \* 2 = 18  potenciar 9 \*\* 2 = 81  radicar 9 \*\* (1/2) = 3.0 |

En estas funciones dos enteros ingresados por el programador

dato\_primero=9

dato\_segundo=2

ya hemos visto como ingresar un objeto string con input, validarlo y hacer un casting de string a entero.

Ahora generaremos en una función este ingreso, validación y casting

|  |
| --- |
| def ingresar\_validar\_cambiar(tipo\_de\_salida, texto\_de\_consulta):  salida=""  while salida == "":  ingreso = input (f"{texto\_de\_consulta} :")  if tipo\_de\_salida==str:  salida = ingreso.title()  elif tipo\_de\_salida==int and ingreso.isdecimal() is True:  salida = int(ingreso)  elif tipo\_de\_salida==float and ingreso.replace(".","",1).isdecimal() is True:  salida = float(ingreso)  else:  continue  return salida  #---------------------------------------------------------------------------------------------  desde\_funcion = ingresar\_validar\_cambiar(str, "ingrese un texto")  print (f"{desde\_funcion=} clase:{type(desde\_funcion)}")  desde\_funcion = ingresar\_validar\_cambiar(int, "ingrese un entero")  print (f"{desde\_funcion=} clase:{type(desde\_funcion)}")  desde\_funcion = ingresar\_validar\_cambiar(float, "ingrese un flotante")  print (f"{desde\_funcion=} clase:{type(desde\_funcion)}") |

Salida esperada por consola

|  |
| --- |
| ingrese un texto :pYTHON eS gENIAL#<-------------------------------ingreso de la altura del usuario  desde\_funcion='Python Es Genial' clase:<class 'str'>  ingrese un entero :8#<-------------------------------------------------------------------------ingreso de la altura del usuario  desde\_funcion=8 clase:<class 'int'>  ingrese un flotante :3.14159#<--------------------------------------------------------ingreso de la altura del usuario  desde\_funcion=3.14159 clase:<class 'float'> |

Las funciones pueden tener parámetros opcionales con valores predeterminados, lo que permite llamar a la función sin proporcionar todos los argumentos.

**Argumentos posicionales y por omisión (palabras clave):**

Los argumentos posicionales se pasan a una función según el orden en el que se definen en la declaración de la función.

La sintaxis básica de argumentos posicionales en Python es la siguiente:

**def** mi\_funcion (parametro1, parametro2):

# parametro1 tendrá el valor de 8

# parametro2 tendrá el valor de 9

return

mi\_funcion (8,9)

Los argumentos de palabras clave se pasan a una función utilizando el nombre del parámetro al que se desea asignar el valor. Esto permite pasar los argumentos en cualquier orden.

La sintaxis básica de argumentos posicionales en Python es la siguiente:

**def** mi\_funcion (parametro1, parametro2):

# parametro1 tendrá el valor de 8

# parametro2 tendrá el valor de 9

return

#--------------------------------------------------------

mi\_funcion (parametro2=9, parametro1=8)

**Parámetros arbitrarios en Python**

Hay dos tipos de parámetros arbitrarios \* y \*\*

Un asterisco \* y recopilan todos los argumentos posicionales en una tupla.

def sumar(\*numeros):

total = sum(numeros)

print("La suma es:", total)

#--------------------------------------------------------

sumar(1, 2, 3, 4, 5)

Dos asteriscos \*\* y recopilan todos los argumentos de palabras clave en un diccionario.

def multiplicar(\*\*diccionario):

for clave, valor in diccionario.items():

print(f"{clave} - {valor} ")

#--------------------------------------------------------

multiplicar( \*\*{"1ro" : 1 , "2do" : 2 , "3ro" : 3 , "4to" : 4, "5to" : 5})

**Funciones con parámetros y operadores**

En Python, el símbolo de asterisco \* se utiliza para desempaquetar argumentos de una secuencia (como una lista o una tupla) y el símbolo de barra diagonal / se utiliza para separar los argumentos posicionales de los argumentos de palabras clave en la definición de una función. Veamos cómo se usan en cada caso:

**Desempaquetar argumentos con \*:**

Si tienes una secuencia de valores y deseas pasarlos como argumentos individuales a una función, puedes utilizar el operador \* para desempaquetar la secuencia.

Esto es útil cuando tienes una lista o una tupla y deseas pasar sus elementos como argumentos a una función que espera argumentos separados.

La sintaxis básica de parámetros tuplas con \* en Python es la siguiente:

def suma(a, b, c):

resultado = a + b + c

print(f"el resultado de la suma es {resultado}")

#--------------------------------------------------------

numeros = [1, 2, 3]

suma(\*numeros) # Equivalente a suma(1, 2, 3)

--------------------------------------------------------------------

def suma(\*valores):

resultado = sum(valores)

print(f"el resultado de la suma es {resultado}")

#--------------------------------------------------------

numeros = [1, 2, 3]

suma(\*numeros) # Equivalente a suma(1, 2, 3)

**Separar argumentos posicionales y de palabras clave con /:**

En la definición de una función, el símbolo / se utiliza para marcar el punto en el que terminan los argumentos posicionales y comienzan los argumentos de palabras clave.

Esto es relevante cuando deseas definir una función que tenga una cantidad fija de argumentos posicionales y luego argumentos opcionales que se pueden especificar mediante palabras clave.

La sintaxis básica de parámetros con / en Python es la siguiente:

def funcion(a, b, /, c, d):

total = a + b + c + d

print(f"La suma es ({a=}+{b=}+{c=}+{d=}):", total)

#--------------------------------------------------------

funcion(1, 2, 3, 4) # Equivalente a funcion(1, 2, c=3, d=4)

funcion(1, 2, d=4 , c=3)

funcion(1, 2, c=3 , d=4)

funcion(b=2, a=1, d=4 , c=3)# no valido

En este ejemplo, a y b son argumentos posicionales, mientras que c y d son argumentos de palabras clave. Al marcar la posición de /, se indica que los primeros dos argumentos son posicionales y los últimos dos argumentos deben especificarse mediante palabras clave.

**Orden de los argumentos**

1. posicionales
2. \* arbitrarios
3. \*\* kwords -clave valor
4. por omisión

**Recursividad**

Se denomina llamada recursiva (o recursividad), a aquellas funciones que en su algoritmo, hacen referencia sí misma (se llama dentro de la misma función a si misma).

Python admite las llamadas recursivas, permitiendo a una función, llamarse a sí misma, de igual forma que lo hace cuando llama a otra función.

La sintaxis básica recursividad en funciones Python es la siguiente:

|  |
| --- |
| def funcion\_rec (parametro):  #contenido de la función  #condición  #cumple con la condicional  return funcion\_rec (parametro\_modificado)  #no cumple con la condicional  return salida\_final    regreso = funcion\_rec (argumento) |

|  |
| --- |
| def factorial\_recursivo(valor):  if valor == 1:  return 1  else:  return valor \* factorial\_recursivo(valor-1)  salida = 10**#< -----------------modificar**  regreso = factorial\_recursivo(salida) |

·

|  |
| --- |
| el factorial de 10 es 3628800 |

·

|  |
| --- |
| def funcion\_password (intento=1):  respuesta = input("Ingrese su password (1234) :")  if respuesta != "1234":  if intento <= 3:  print (f"\nError {intento}! Inténtalo de nuevo" )  intento += 1  funcion\_password(intento) # Llamada recursiva  else:  print ("Ingreso no permitido" )  exit()  else:  print ("Ingresaste!!!!!!" )  funcion\_password() |

·

|  |
| --- |
| Ingrese su password (1234) :**4321#<-------------------------------ingreso de la altura del usuario**  Error 1! Inténtalo de nuevo  Ingrese su password (1234) :0000#<-------------------------------ingreso de la altura del usuario  Error 2! Inténtalo de nuevo  Ingrese su password (1234) :9876#<-------------------------------ingreso de la altura del usuario  Error 3! Inténtalo de nuevo  Ingrese su password (1234) :1212#<-------------------------------ingreso de la altura del usuario  Ingreso no permitido |

·

|  |
| --- |
| Ingrese su password (1234) :1234#<-------------------------------ingreso de la altura del usuario  Ingresaste!!!!!! |

**Anotaciones en funciones**

Existe una funcionalidad relativamente reciente en Python llamada function annotation o anotaciones en funciones. Dicha funcionalidad nos permite añadir metadatos a las funciones, indicando los tipos esperados tanto de entrada como de salida.

La sintaxis básica anotación en funciones Python es la siguiente:

def funcion\_anot (ingreso : int)-> float:

#contenido de la función

#se espera el ingreso como entero

#cumple con la condicional

return ingreso/2# siempre una división regresa un float

#--------------------------------------------------------

salida = 5

regreso = funcion\_anot (salida)

print (f"el e {salida} es {regreso} ")

Las anotaciones son muy útiles de cara a la documentación del código, pero no imponen ninguna norma sobre los tipos. Esto significa que se puede llamar a la función con un parámetro que no sea int, y no obtendremos ningún error.

|  |
| --- |
| def fibonacci\_recursivo(valor):  if valor == 0:  return 0  elif valor == 1:  return 1  else:  return fibonacci\_recursivo(valor-1) + fibonacci\_recursivo(valor-2)  salida = 10  regreso = fibonacci\_recursivo(salida)  print (f"el fibonacci de {salida} es {regreso} ") |

·

|  |
| --- |
| el fibonacci de 10 es 55 |

**docstrings**

**Help! I need somebody**

**Help! Not just anybody**

**Help! You know I need someone**

**Help!**

Writer(s): John Lennon, Paul Mccartney

docstrings son cadenas de texto que se utilizan para documentar la función y explicar su propósito, parámetros, comportamiento y demás información.

La sintaxis básica de docstrings en Python es la siguiente:

**def** mi\_funcion (parametro):

"""

Este es el docstrings de "mi\_funcion"

mi\_funcion (parametro):

requiere un argumentos:

retorna el argumento \*5

si el argumento es numérico multiplica en valor por 5

si el argumento es string replica este 5 veces

"""

**return** parametro\*5

#---------------------------------------------------------------------

print(mi\_funcion.\_\_doc\_\_)

help(mi\_funcion)

print (f"{mi\_funcion(5)=}")

print (f"{mi\_funcion(' cinco ')=}")

|  |
| --- |
| def ingresar\_validar\_cambiar(tipo\_de\_salida, texto\_de\_consulta):  """  Este es el docstrings de la función  la funcion ingresar\_validar\_cambiar (tipo\_de\_salida, texto\_de\_consulta)  requiere dos argumentos:  tipo\_de\_salida soporta tipo str, int y float  texto\_de\_consulta un string con el mensaje para el usuario que explique el dato  que se espera  """  salida=""  while salida == "":  ingreso = input (f"{texto\_de\_consulta} :")  if tipo\_de\_salida==str:  salida = ingreso.title()  elif tipo\_de\_salida==int and ingreso.isdecimal() is True:  salida = int(ingreso)  elif tipo\_de\_salida==float and ingreso.replace(".","",1).isdecimal() is True:  salida = float(ingreso)  else:  continue  return salida    print(ingresar\_validar\_cambiar.\_\_doc\_\_)  help(ingresar\_validar\_cambiar)  desde\_funcion = ingresar\_validar\_cambiar(str, "ingrese un texto")  print (f"{desde\_funcion=} clase:{type(desde\_funcion)}")  desde\_funcion = ingresar\_validar\_cambiar(int, "ingrese un entero")  print (f"{desde\_funcion=} clase:{type(desde\_funcion)}")  desde\_funcion = ingresar\_validar\_cambiar(float, "ingrese un flotante")  print (f"{desde\_funcion=} clase:{type(desde\_funcion)}") |

Salida esperada por consola

|  |
| --- |
| #print(ingresar\_validar\_cambiar.\_\_doc\_\_)  Este es el docstrings de la función  la funcion ingresar\_validar\_cambiar (tipo\_de\_salida, texto\_de\_consulta)  requiere dos argumentos:  tipo\_de\_salida soporta tipo str, int y float  texto\_de\_consulta un string con el mensaje para el usuario que explique el dato  que se espera  #help(ingresar\_validar\_cambiar)  Help on function ingresar\_validar\_cambiar in module \_\_main\_\_:  ingresar\_validar\_cambiar(tipo\_de\_salida, texto\_de\_consulta)  Este es el docstrings de la función  la función ingresar\_validar\_cambiar (tipo\_de\_salida, texto\_de\_consulta)  requiere dos argumentos:  tipo\_de\_salida soporta tipo str, int y float  texto\_de\_consulta un string con el mensaje para el usuario que explique el dato  que se espera  ingrese un texto :…………………... |

**Retorno**

**1) Retorno único:**

En algunos lenguajes de programación cada función tiene solamente un return. En python no es asi. Aunque la comunidad prefiere que solo se coloque uno.

Por otra parte en breve con try except veremos que el return puede ser interceptado y descartado.

La sintaxis básica de retorno único o múltiple:

**def** mi\_funcion (parametro):

if parametro.isdecimal():

return int(parametro)

else:

return False

#---------------------------------------------------------------------

print (f"{mi\_funcion(5)=}")

print (f"{mi\_funcion(' cinco ')=}")

**2) Empaquetados y desempaquetados.**

Cuando es necesario retornar mas de un objeto al llamamiento de la función se retorna separados por comas, esto genera una tupla, no obstante podemos retornar diccionarios y otros objetos

La sintaxis básica de empaquetados y desempaquetados en Python es la siguiente:

def mi\_funcion (parametro):

if isinstance(parametro, str):

if parametro.isdigit() and parametro < 18:

return int(parametro), "Ud no puede ingresar al bar"

elif parametro.isdigit() and parametro >= 18:

return int(parametro), "Bienvenido, Ud puede beber alcohol"

elif isinstance(parametro, (int,float)):

if parametro < 18:

return parametro, "Ud no puede ingresar al bar"

elif parametro >= 18:

return parametro, "Bienvenido, Ud puede beber alcohol"

else:

return None

#---------------------------------------------------------------------

print (f"{mi\_funcion(30)=}")

print (f"{mi\_funcion(15)=}")

print (f"{mi\_funcion(' quince ')=}")

La sintaxis básica de empaquetados y desempaquetados en Python es la siguiente:

def mi\_funcion ():

return "Python","es genial"

#---------------------------------------------------------------------

regreso = mi\_funcion()

print (f"{regreso=}")

regreso\_0, regreso\_1 = mi\_funcion()

print (f"primer valor de regreso {regreso\_0=}")

print (f"segundo valor de regreso {regreso\_1=}")