Módulo 9:

**Funciones propias**:

* 1. def return
  2. Argumentos posicionales y por omisión
  3. Parámetros arbitrarios en Python
  4. Funciones con parámetros y operadores
  5. Orden de los argumentos
  6. Recursividad
  7. Anotaciones en funciones
  8. docstrings
  9. Retorno Empaquetados y desempaquetados.
  10. 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 |

Las funciones en Python son esenciales para la creación de código modular, reutilizable y mantenible.

Una función en programación es un componente fundamental que consiste en un bloque de código encapsulado diseñado para realizar una tarea específica de manera independiente.

Las funciones son esenciales en el desarrollo de software debido a que facilitan la reutilización de código y promueven la modularidad, eliminando la necesidad de repetir secciones de código innecesariamente en diferentes partes de un programa.

Cuando una acción particular se requiere con frecuencia en un programa, es una indicación clara de que esa acción debería ser encapsulada en una función para su posterior reutilización. Además, las funciones permiten una mejor organización y estructuración del código, lo que facilita su mantenimiento y comprensión.

|  |
| --- |
| Uno no define la función **print** como salida por consola del dato que este entre paréntesis.  print ("hola mundo") |

1. Definición y estructura básica de una función

* **Funciones built-in (o funciones integradas):**

Estas son funciones que ya vienen incluidas en Python y están listas para ser usadas sin necesidad de que el programador las defina o importe ningún módulo adicional. Son parte de la biblioteca estándar de Python y cubren operaciones comunes que se utilizan con frecuencia.

* **print():** para mostrar datos en pantalla.
* **len():** para obtener la longitud de un objeto (como una lista o una cadena).
* **type():** para obtener el tipo de un objeto.
* **sum():** para sumar los elementos de un iterable.
* **enumerate():** para obtener un iterable que contiene tuplas con índices y valores.
* **Funciones propias (o funciones definidas por el usuario):**

Una función en programación, y en particular en Python, es un bloque de código reutilizable que realiza una tarea específica. Las funciones están diseñadas para aceptar entradas (argumentos), procesarlas, y producir una salida (resultado), o ejecutar una serie de instrucciones sin devolver ningún valor.

1. Son las funciones que tú mismo defines en tu código para realizar tareas específicas.
2. Función es un objeto con nombre propio con un bloque de código reutilizable.
3. Este bloque se utiliza para realizar una tarea específica y permiten dividir el código en fragmentos más pequeños y lógicos.
4. Sintaxis:
   * + 1. **def:** Esta palabra clave indica a Python que una nueva función está siendo definida. Es un elemento fundamental en la creación de funciones.
       2. **Nombre de la función**: Tras def, se coloca el nombre de la función. Este debe ser válido y seguir las reglas de nomenclatura de Python:
       - Debe comenzar con una letra o un guion bajo.
       - Puede incluir números, pero no comenzar con ellos.
       - Los nombres suelen escribirse en minúsculas y se recomienda usar guiones bajos para separar palabras, siguiendo la convención de estilo snake\_case.
       - No puedes usar palabras reservadas de Python (como if, for, while, etc.).
       1. **Paréntesis** (): Después del nombre de la función, se colocan los paréntesis, dentro de los cuales se pueden definir **parámetros** (valores que la función espera recibir). Puede haber cero, uno o más parámetros, separados por comas. Los parámetros también pueden tener **valores predeterminados**, lo que significa que si no se proporcionan argumentos en la llamada a la función, estos valores serán utilizados.
       2. **Dos puntos** :: Es obligatorio cerrar la línea de la definición de la función con dos puntos :. Este símbolo indica el comienzo del bloque de código que pertenece a la función.
       3. **Indentación**: La siguiente línea debe estar indentada, lo cual es obligatorio en Python para definir qué código pertenece al bloque de la función. Generalmente, la indentación se realiza con cuatro espacios o con la tecla Tab.
       4. **Cuerpo de la función**: El cuerpo o contenido de la función incluye las instrucciones que Python ejecutará cuando la función sea llamada. Este código representa las acciones que la función realizará usando los parámetros proporcionados.
       5. return **(opcional)**: Al final del bloque de la función, se puede incluir la declaración return. Esta instrucción devuelve un valor específico y finaliza la ejecución de la función. Si no se llama a return, la función devolverá None.
       - Ten presente que si olvidas el paréntesis() o los dos puntos : cuando intentes definir una nueva función, Python automáticamente te responderá con un SyntaxError.
       - 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)
5. Estos bloques o funciones permiten la organización, la reutilización y el mantenimiento del código.
6. Se puede asignar a un encargado (persona grupo) a cada función , permitiendo dividir y organizar y responsabilizar las tareas.
7. Una función no es más que un bloque de código aislado que lleva a cabo una tarea específica.
8. Las funciones son muy útiles en programación debido a que eliminan los innecesarios y excesivos ¨copia y pega¨ de código a través de un programa.
9. Si una acción específica es requerida a menudo en tu código, es un buen indicador de la necesidad de escribir una función. Las funciones son para ser reutilizadas.
10. Las funciones también te ayudan a organizar tu código.
11. Si necesitas hacer un cambio en tu programa, solo tendrás que actualizar la declaración de la función concerniente. Esto evita que tengas que buscar diferentes piezas del mismo código, esparcido en diferentes campos de tu programa.
12. Esto cumple con el principio de desarrollo de software DRY (Don't Repeat Yourself), que significa; no te repitas.
13. El código dentro de una función se ejecuta solo cuando la función es llamada.
14. Las funciones pueden aceptar argumentos y valores predeterminados, pudiendo o no regresar otros
15. Los argumentos son los datos que se pasan a una función cuando se llama. Los parámetros son las variables que la función define para recibir esos datos y utilizarlos en su ejecución. En Python, los parámetros no tienen un tipo de dato predefinido, sino que su tipo lo determina el valor del argumento que se les pase al llamar la función. Esto se debe a que Python es un lenguaje de tipado dinámico.
16. En Python, los recursos y datos generados dentro de una función que no se retornan son, en su mayoría, eliminados o liberados automáticamente cuando la función termina su ejecución, liberando esos recursos para que el sistema los reutilice.

Python utiliza un sistema de gestión de memoria automático, llamado recolección de basura (garbage collection), que se encarga de liberar la memoria que ya no está en uso.  
Cuando una función es llamada, se crea un nuevo espacio de memoria llamado entorno local o alcance local (local scope), donde se almacenan las variables y datos generados dentro de la función.

Al terminar la ejecución de la función, las variables que fueron creadas en su entorno local se destruyen o se liberan, siempre que no hayan sido retornadas o capturadas por un código externo (por ejemplo, asignadas a una variable fuera de la función).

Esto significa que si la función no devuelve un valor explícito mediante return, los datos locales creados en ella desaparecen una vez que la función finaliza su ejecución.

|  |
| --- |
| def nombre\_funcion(parametro1, parametro2):  # cuerpo de la función  resultado = parametro1 + parametro2  return resultado  regreso = nombre\_funcion(3,7) |

**1. Características principales de una función:**

* **Encapsulamiento:** Una función encapsula código que puede ser utilizado varias veces, evitando la repetición.
* **Modularidad:** Permiten dividir un programa en partes o módulos, lo que hace que sea más fácil de entender, mantener y probar.
* **Reutilización:** Una vez que has definido una función, puedes invocarla cuantas veces necesites a lo largo del programa.
* **Entradas (parámetros):** Las funciones pueden aceptar valores de entrada (llamados argumentos o parámetros), que son variables que la función utiliza para realizar su tarea.
* **Salida (valor de retorno):** Las funciones pueden devolver un valor como resultado de su ejecución utilizando la palabra clave return.
* **Abstracción:** Al usar una función, no necesitas conocer los detalles internos de su implementación, solo cómo utilizarla (qué argumentos pasarle y qué tipo de resultado esperar). Esto mejora la claridad del código.

**2. Argumentos posicionales y por omisión**

En Python, los argumentos pueden ser pasados por posición o por omisión:

* **Argumentos posicionales**: son aquellos en los que el valor pasado a la función depende del orden en que aparecen.
* **Argumentos por omisión**: permiten definir un valor predeterminado para un parámetro, que se usará si no se pasa un argumento correspondiente. Esto es útil para crear funciones más flexibles.

|  |
| --- |
| def saludar(nombre, mensaje="Hola"):  print(f"{mensaje}, {nombre}")  saludar("Juan") # Imprime: "Hola, Juan"  saludar("Ana", "Buen día") # Imprime: "Buen día, Ana" |

**3. Parámetros arbitrarios en Python**

Si no sabes cuántos parámetros va a recibir una función, puedes utilizar parámetros arbitrarios con \*args (para una cantidad variable de argumentos posicionales) o \*\*kwargs (para una cantidad variable de argumentos con nombre):

|  |
| --- |
| def sumar(\*args):  return sum(args) |
| def info(\*\*kwargs):  for clave, valor in kwargs.items():  print(f"{clave}: {valor}") |

**4. Funciones con parámetros y operadores**

Las funciones también pueden contener operadores dentro de sus parámetros. Un ejemplo común es establecer valores predeterminados para los parámetros basados en operaciones:

|  |
| --- |
| def multiplicar(x, y=2):  return x \* y  resultado = multiplicar(5) # Devuelve 10, ya que y=2 por defecto |

**5. Orden de los argumentos**

El orden de los argumentos en una función es importante cuando mezclamos posicionales, por omisión, \*args y \*\*kwargs. El orden correcto es:

En Python, los parámetros en las funciones se deben ordenar según ciertas reglas para asegurar una correcta interpretación por el intérprete. El orden es:

1. **Parámetros Posicionales (obligatorios)**: Son los argumentos que la función espera recibir sin nombre específico y en un orden determinado. Por ejemplo, en def funcion(a, b):, a y b son parámetros posicionales.
2. **Parámetros con valores predeterminados (default)**: Son los parámetros que tienen un valor por defecto, lo cual hace que sean opcionales. Se colocan después de los posicionales. Ejemplo: def funcion(a, b, c=10):.
3. \*args **(Argumentos Variables Posicionales)**: Esta sintaxis permite recibir una cantidad arbitraria de argumentos posicionales y los agrupa en una tupla. Se usa el símbolo \* seguido de un nombre de variable (por convención, args), y debe ubicarse después de los parámetros posicionales y los predeterminados. Ejemplo: def funcion(a, b, c=10, \*args):.
4. **Parámetros con nombre solo (Keyword-only arguments)**: Estos parámetros deben proporcionarse por nombre y no por posición. En Python, se puede especificar que un parámetro sea de tipo keyword-only al colocar \* antes del parámetro en cuestión. Ejemplo: def funcion(a, b, \*, d):, donde d es obligatorio y debe pasarse con el nombre d=valor.
5. \*\*kwargs **(Argumentos Variables con Nombre)**: Similar a \*args, pero para argumentos con nombre. Usando \*\* seguido de un nombre de variable (por convención, kwargs), se agrupan en un diccionario. Deben ir al final de todos los parámetros. Ejemplo: def funcion(a, b, c=10, \*args, d, \*\*kwargs).

|  |
| --- |
| **Ejemplo completo** |
| def ejemplo(a, b, c=10, \*args, d, \*\*kwargs):  print("a:", a)  print("b:", b)  print("c:", c)  print("args:", args)  print("d:", d)  print("kwargs:", kwargs) |
| ejemplo(1, 2, 20, 30, 40, d=50, extra1="test", extra2="test2") |

.

Este orden permite que cada tipo de parámetro se gestione de manera predecible en el código.

1. / **(Parámetros Posicionales Solo)**: Los parámetros que aparecen antes del símbolo / solo se pueden pasar por posición, no por nombre. Esto es útil cuando se quieren evitar ambigüedades en la llamada a la función.
2. **Parámetros Posicionales (obligatorios)**: Son los argumentos que la función espera recibir sin nombre específico y en un orden determinado. Ejemplo: def funcion(a, b):.
3. **Parámetros con valores predeterminados (default)**: Son parámetros opcionales que tienen un valor por defecto y se colocan después de los posicionales obligatorios. Ejemplo: def funcion(a, b, c=10):.
4. \* **(Separador de Keyword-only Arguments)**: El símbolo \* indica que los parámetros que aparecen después de él solo se pueden pasar por nombre, y no por posición. Esto ayuda a evitar errores en funciones con muchos parámetros opcionales o parámetros que se desean pasar únicamente por nombre. Ejemplo: def funcion(a, b, \*, d):.
5. \*args **(Argumentos Variables Posicionales)**: Agrupa cualquier cantidad arbitraria de argumentos posicionales adicionales en una tupla. Debe ir después de los parámetros posicionales y predeterminados. Ejemplo: def funcion(a, b, c=10, \*args):.
6. **Parámetros Keyword-only**: Argumentos que deben pasarse explícitamente por nombre debido a la presencia de \* o \*args. Ejemplo: def funcion(a, b, c=10, \*, d):.
7. \*\*kwargs **(Argumentos Variables con Nombre)**: Agrupa cualquier cantidad arbitraria de argumentos con nombre adicionales en un diccionario. Debe ubicarse al final. Ejemplo: def funcion(a, b, c=10, \*args, d, \*\*kwargs):.

|  |
| --- |
| **Ejemplo Completo nuevo** |
| def ejemplo(a, b, /, c=10, \*, d, e=20, \*\*kwargs):  print("a:", a)  print("b:", b)  print("c:", c)  print("d:", d)  print("e:", e)  print("kwargs:", kwargs) |
| Llamada válida:  ejemplo(1, 2, c=15, d=25, e=30, extra1="test", extra2="test2") |

Explicación del orden completo:

* **Posicionales obligatorios (**/**)**: solo se pueden pasar por posición.
* **Parámetros predeterminados**: tienen valores opcionales.
* \* **(Keyword-only arguments)**: a partir de aquí, los parámetros solo pueden pasarse con nombre.
* \*args **y** \*\*kwargs: para argumentos variables posicionales y con nombre.

**6. Recursividad**

La **recursividad** ocurre cuando una función se llama a sí misma. Es útil en problemas que pueden dividirse en subproblemas más pequeños, como la factorización o la búsqueda en árboles.

Ejemplo clásico de función recursiva para calcular el factorial de un número:

|  |
| --- |
| def factorial(n):  if n == 1:  return 1  else:  return n \* factorial(n-1) |

Es importante manejar los casos base correctamente en la recursividad para evitar bucles infinitos.

**7. Anotaciones en funciones**

Las **anotaciones** son una característica opcional que permite agregar información sobre los tipos de los parámetros y el valor de retorno de una función, sin afectar su comportamiento:

|  |
| --- |
| def sumar(a: int, b: int) -> int:  return a + b |

Esto no impone restricciones, pero es útil para documentar y mejorar la legibilidad del código.

**8. Docstrings**

Los **docstrings** son una manera de documentar funciones en Python. Se colocan justo después de la definición de la función y deben seguir un formato estandarizado (como el de PEP 257).

|  |
| --- |
| def sumar(a, b):  """  Suma dos números.  Parámetros:  a -- Primer número  b -- Segundo número  Retorna:  La suma de a y b  """  return a + b |

Esto es útil cuando otros desarrolladores necesitan entender lo que hace la función.

**9. Retorno empaquetado y desempaquetado**

Las funciones en Python pueden devolver múltiples valores empaquetados como tuplas, que luego pueden ser desempaquetados:

|  |
| --- |
| def retornar\_varios():  return 1, 2, 3  a, b, c = retornar\_varios() # Desempaqueta la tupla en variables |

**10. Ámbitos de los distintos objet**os

El **ámbito** (o **scope**) de una variable determina dónde puede ser accedida en el programa. Las variables definidas dentro de una función tienen un ámbito local y no son accesibles fuera de ella. Python sigue las reglas LEGB (Local, Enclosing, Global, Built-in) para la resolución de nombres.

**11. Empaquetado de entrada y salida**

Python permite empaquetar múltiples argumentos usando \* para colecciones como listas o tuplas, y \*\* para diccionarios:

|  |
| --- |
| def sumar(\*args):  return sum(args)  numeros = [1, 2, 3]  print(sumar(\*numeros)) # Desempaqueta la lista |

De manera similar, los resultados de una función pueden empaquetarse al ser retornados como una tupla.

|  |
| --- |
| 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) and parametro.replace(".","").isdigit() :  parametro = float(parametro)  if isinstance(parametro, (int,float)):  if parametro < 18:  return parametro, "Ud no puede ingresar al bar"  else:# 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=}") |

.

|  |
| --- |
| def suma(a, \*, b, c):  return a + b + c  Desglose: a: Este parámetro puede ser pasado por posición. b y c: Son parámetros solo por palabra clave debido al asterisco \*. Esto significa que deben pasarse obligatoriamente con su nombre cuando se llame a la función.  # Llamada válida, pasando `a` por posición, y `b` y `c` por palabra clave resultado = suma(1, b=2, c=3) print(resultado) # Salida: 6  # Otra llamada válida, con diferentes valores resultado = suma(10, b=20, c=30) print(resultado) # Salida: 60  Llamadas inválidas: No puedes pasar b o c por posición, solo se pueden pasar por nombre: # Llamada inválida: `b` y `c` deben ser pasados por nombre # resultado = suma(1, 2, 3) # Error: TypeError  Resumen: El primer argumento a debe ser pasado por posición. Los argumentos b y c deben ser pasados por palabra clave. |

En este caso hemos añadido el asterisco (\*) en la sección de parámetros, le indicamos a Python que todos los parámetros que se encuentren a la derecha del asterisco únicamente podrán tomar su valor por nombre y no por posición.

3).Positional only

A partir de la versión 3.8 de Python, es posible la asignación de valores solo por posición, es decir, lo contrario de lo que acabamos de ver. Y para ello haremos uso del slash (/).

|  |
| --- |
| def suma(a, b, /, c):  return a + b + c  # Llamada con todos los parámetros por posición resultado = suma(1, 2, 3) print(resultado) # Imprime: 6  # Llamada con 'c' pasado por palabra clave resultado = suma(1, 2, c=3) print(resultado) # Imprime: 6  Llamada inválida: # Intento de pasar 'a' o 'b' por palabra clave (esto causará un error) resultado = suma(a=1, b=2, c=3) # Error: 'a' y 'b' son solo posicionales |

Para este nuevo ejemplo, le indicamos a Python que todos los parámetro que se encuentren a la izquierda del slash , solo podrán tomar su valor por posición y no por nombre.

|  |
| --- |
| def function(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2):  \\_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_\_\_/  | | | \_\_Keyword only  | |  | | \_\_Positional or keyword  |  | \_\_Positional only |

Parámetros de número variable: \*args y \*\*kwargs en Python.

Hasta aquí te he enseñado lo fundamental sobre los distintos tipos de parámetros al definir una función Python. No obstante, si quieres ser un auténtico pythonista  y completar tu formación, debes leer el siguiente tutorial para entender no solo como parámetros, sino que también pueden ser pasados como argumentos . En este post quiero explicarte qué significan y cuándo usarlos.. Siguiendo este tutorial, habrás completado tu formación sobre los tipos de parámetros en una función Python.

Lo que realmente indica que el parámetro es de este tipo es el símbolo ‘\*’, el nombre args se usa por convención.

* El parámetro recibe los argumentos como una tupla.
* Es un parámetro opcional. Se puede invocar a la función haciendo uso del mismo, o no.
* El número de argumentos al invocar a la función es variable.
* Son parámetros posicionales, por lo que, a diferencia de los parámetros con nombre, su valor depende de la posición en la que se pasen a la función.

En Python, el concepto de **"positional-only" (solo posicional)** se refiere a la posibilidad de definir parámetros que **solo pueden ser pasados por posición** y no mediante el nombre del argumento. Esta funcionalidad fue introducida en Python 3.8 mediante la sintaxis de la barra / (slash) en la definición de funciones.

**4). Parámetros **solo posicionales** (/)**

* Los parámetros que aparecen **antes de la barra** / en la definición de una función solo pueden recibir valores cuando se llaman por posición y no por nombre.
* Esto significa que cuando llames a una función con parámetros definidos antes de la /, **no puedes usar su nombre como argumento**, solo puedes pasar el valor en la posición correspondiente.

**Ejemplo de parámetros **solo posicionales**:**

|  |
| --- |
| def funcion(pos1, pos2, /):  print(f"pos1: {pos1}, pos2: {pos2}")  # Llamadas válidas  funcion(10, 20) # OK  # Llamadas inválidas  funcion(pos1=10, pos2=20) # Error: los argumentos solo pueden ser pasados por posición |

* En este ejemplo, pos1 y pos2 son parámetros que solo pueden recibir sus valores por posición. No es posible invocarlos por nombre, como pos1=10 o pos2=20.
* Parámetros **mixtos** (/ y \*)

En la definición de la función que mencionas:

|  |
| --- |
| def function(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2):  pass |

* pos1 **y** pos2 son parámetros **solo posicionales** debido a la barra /. Deben ser pasados exclusivamente por su posición en la llamada a la función.
* pos\_or\_kwd es un parámetro que puede ser pasado tanto **por posición** como **por palabra clave** (nombre del argumento). Está definido **después de la barra** / pero **antes del asterisco** \*, por lo que tiene ambas opciones.
* kwd1 **y** kwd2 son parámetros que deben ser pasados **únicamente por palabra clave** debido al asterisco \*, lo que significa que siempre deben ser referidos por su nombre al llamar a la función.

Ejemplo explicando cada tipo de parámetro:

|  |
| --- |
| def function(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2):  print(f"""  pos1: {pos1},  pos2: {pos2},  pos\_or\_kwd: {pos\_or\_kwd},  kwd1: {kwd1},  kwd2: {kwd2}""")  # Llamada válida  function(1, 2, 3, kwd1=4, kwd2=5)  # Llamada inválida: 'pos1' y 'pos2' no pueden ser pasados por nombre  # function(pos1=1, pos2=2, pos\_or\_kwd=3, kwd1=4, kwd2=5) # Error |

Desglose de la llamada válida:

* 1 **y** 2: Se asignan a los parámetros **solo posicionales** pos1 y pos2. No se puede hacer referencia a ellos por nombre, solo se pasan por posición.
* 3: Se pasa como valor del parámetro pos\_or\_kwd, que puede ser pasado por posición o por palabra clave.
* kwd1=4, kwd2=5: Estos parámetros deben ser **pasados por nombre** porque están después del asterisco \*, lo que los convierte en **solo palabra clave**.

¿Por qué usar parámetros solo posicionales?

1. **Compatibilidad**: Si quieres que una API mantenga la compatibilidad con versiones anteriores, puedes asegurar que ciertos parámetros solo puedan pasarse por posición y no por nombre.
2. **Claridad**: Evita que los usuarios de la función confundan el nombre del parámetro con algún otro uso o término específico del contexto.
3. **Rendimiento**: En algunos casos, el manejo de parámetros por posición puede ser más eficiente que por nombre.

Resumen:

* **Parámetros antes de** /: Solo pueden pasarse por **posición** (positional-only).
* **Parámetros entre** / **y** \* : Pueden pasarse por **posición o palabra clave** (mixtos).
* **Parámetros después de** \* : Solo pueden pasarse por **palabra clave** (keyword-only).