**Módulo 4**

**4-Paradigma funcional**

**¿Qué es el paradigma funcional?**

**¿Por qué utilizar el paradigma funcional en Python?**

**El espacio de nombres (namespace)**

1. Python es un lenguaje de muy alto nivel en el que todos sus elementos son objetos, incluyendo los tipos de datos básicos.
2. La gestión del uso de la memoria es automático en Python, tanto para la asignación de memoria al crear un objeto, como para la recuperación de memoria al desecharlo (**garbage collector** )
3. El espacio de nombres (namespace) contiene un listado de los objetos existentes en la memoria del sistema y los nombres a los que están ligados.
4. Un objeto puede tener estar asociado a mas de una etiqueta o nombre ( Principalmente observar alias en colecciones).
5. Si un objeto no está ligado al menos a un nombre, dicho elimina de memoria.
6. El intérprete de Python tiene un espacio de nombres principal, pero cada función, módulo y objeto tiene su propio espacio de nombres.

**Comparación con el paradigma imperativo y el paradigma orientado a objetos**

**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.

**Características claves de las funciones en Python**

* Modularidad
* Reutilización de código
* Encapsulación
* Parámetros
* Retorno de valores
* Ámbito (scope) de las variables

**Ámbitos o Scope**

* Ámbito global (Global scope)
* Ámbito de función (Function scope)
* Ámbito de bloque (Block scope)

**Definiendo nombres en el ámbito global con la expresión global.**

**Arquitectura de una función.**

**Parámetros y argumentos**

* Argumentos posicionales
* Argumentos por nombre del parámetro (palabra clave)
* Parámetros con argumentos por defecto u omisión.
* Parámetros arbitrarios en Python
* Funciones con parámetros y operadores
* Desempaquetar argumentos con \*
* Separar argumentos posicionales y de palabras clave con /

**Orden de los argumentos**

* posicionales
* \* arbitrarios
* \*\* kwords -clave valor
* por omisión

**Recursividad**

**Anotaciones en funciones** (function annotation)

**Manejo de documentación de funciones - docstrings**

**Retorno desde funciones**

* Retorno único
* Empaquetados y desempaquetados.

**Funciones anidadas.**

**Funciones lambda**

**Funciones de orden superior.**

* map
* filter
* reduce (import functools)

**Funciones lambda + funciones de orden superior.**

* lambda + map
* lambda + filter
* lambda + reduce

**Cuadro sinóptico lambda**

**Cuadro sinóptico de Ámbitos en Python**

**Funciones como argumento de entrada a otra función**

**Tabla de punteo de palabras reservadas y funciones incorporadas**

* Ámbito Local (Local Scope):
* Ámbito Encerrado (Enclosing Scope)
* Ámbito Global (Global Scope)
* Ámbito de Nombre Integrado (Built-in Scope)

**4-Paradigma funcional**

**¿Qué es el paradigma funcional?**

El paradigma funcional es un enfoque de programación que se basa en el uso de funciones puras y la eliminación de efectos secundarios. En este enfoque, las funciones son vistas como unidades autónomas que aceptan entradas y producen salidas, sin modificar ningún estado interno ni causar cambios en el ambiente.

El paradigma funcional se enfoca en la resolución de problemas a través de la aplicación de funciones y en la creación de programas más simples, legibles y fáciles de mantener. En lugar de enfocarse en cómo se realizan las cosas, se enfoca en qué se quiere lograr.

¿Por qué utilizar el paradigma funcional en Python?

Python es un lenguaje de programación que soporta múltiples paradigmas, incluyendo el paradigma funcional. Al utilizar este enfoque en Python, podemos aprovechar las ventajas que ofrece, como por ejemplo:

**Código más conciso y legible:** las funciones en el paradigma funcional son más simples y fáciles de entender que en otros enfoques.

Evita efectos secundarios: los efectos secundarios son uno de los principales problemas que dificultan el desarrollo de software, y el paradigma funcional los evita.

Facilita la programación concurrente: el paradigma funcional ofrece una forma natural de escribir código que se adapta muy bien a la programación concurrente.

Comparación con el paradigma imperativo y el paradigma orientado a objetos

El paradigma funcional se diferencia del paradigma imperativo en que se enfoca en "lo que" se quiere hacer, en lugar de "cómo" se hace. En este enfoque, las funciones son unidades independientes y autónomas que no interactúan con el ambiente.

Por otro lado, el paradigma orientado a objetos se enfoca en la creación de objetos que interactúan entre sí a través de métodos. En este enfoque, los objetos encapsulan datos y comportamientos, y los métodos modifican el estado interno del objeto.

En resumen, el paradigma funcional ofrece una forma diferente de abordar la programación, centrándose en la creación de funciones independientes y autónomas que no modifican el ambiente ni causan efectos secundarios. En los siguientes módulos de la clase, profundizaremos en las diferentes herramientas y técnicas que podemos utilizar para aplicar este enfoque en Python.

Las funciones son piezas de código delimitadas y a las que se les puede asignar un nombre con el que pueden ser invocadas. Las funciones son uno de los objetos Python invocables (callable)

La programación funcional es un paradigma de programación que se centra en tratar las operaciones y cálculos como evaluaciones de funciones matemáticas y evita cambiar los estados y las variables mutables. Python es un lenguaje que admite programación funcional y proporciona herramientas y conceptos que permiten escribir código funcional de manera efectiva.

|  |
| --- |
| Malas frases para recordar (lo se son pésimas):  Divide y vencerás.  Cuanto mas chico es el bocado mas rápido se digiere.  Prepara tu propio circo y diviértete por siempre. |

**Funciones como objetos de primera clase:** En Python, las funciones son tratadas como objetos de primera clase, lo que significa que puedes asignarlas a variables, pasarlas como argumentos a otras funciones y devolverlas como valores de otras funciones.

El espacio de nombres (namespace):

****Nota:**** Para fines de claridad, los términos "nombre" e "identificador" serán considerados como sinónimos.

|  |
| --- |
| 1. Python es un lenguaje de muy alto nivel en el que todos sus elementos son objetos, incluyendo los tipos de datos básicos. 2. La gestión del uso de la memoria es automático en Python, tanto para la asignación de memoria al crear un objeto, como para la recuperación de memoria al desecharlo (**garbage collector** ) 3. El espacio de nombres (namespace) contiene un listado de los objetos existentes en la memoria del sistema y los nombres a los que están ligados. 4. Un objeto puede tener estar asociado a mas de una etiqueta o nombre ( Principalmente observar alias en colecciones). 5. Si un objeto no está ligado al menos a un nombre, dicho elimina de memoria. 6. El intérprete de Python tiene un espacio de nombres principal, pero cada función, módulo y objeto tiene su propio espacio de nombres. |

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.

Comparación con el paradigma imperativo y

el paradigma orientado a objetos

El paradigma funcional se diferencia del paradigma imperativo en que se enfoca en "lo que" se quiere hacer, en lugar de "cómo" se hace. En este enfoque, las funciones son unidades independientes y autónomas que no interactúan con el ambiente.

Por otro lado, el paradigma orientado a objetos se enfoca en la creación de objetos que interactúan entre sí a través de métodos. En este enfoque, los objetos encapsulan datos y comportamientos, y los métodos modifican el estado interno del objeto.

En resumen, el paradigma funcional ofrece una forma diferente de abordar la programación, centrándose en la creación de funciones independientes y autónomas que no modifican el ambiente ni causan efectos secundarios. En los siguientes módulos de la clase, profundizaremos en las diferentes herramientas y técnicas que podemos utilizar para aplicar este enfoque en Python.

**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.

Características claves 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.

Ámbitos o Scope

En Python, los ámbitos o scopes determinan la accesibilidad y visibilidad de los distintos tipos de objetos en diferentes partes del código.

Un objeto en ciertos ámbitos puede ser modificado reescrito, con funciones o con métodos de los objetos - Accesibilidad total.

Un objeto en ciertos ámbitos solo puede mostrar su contenido – Visibilidad y utilizar solo los métodos y atributos que no modifica el objeto.

Un objetos con un solo dato creado en la raiz ( root - sobre el margen) tipo str, int, float, bool, puede ser leídos y no modificados si no se les marca como globales.

Un objetos múltiple – Colección con un conjunto de datos creado en la raiz ( root - sobre el margen) tipo list, set, dict,etc (no tuplas o frozensets), puede ser leídos y modificados sin necesidad de marcarlo como globales.

Si los objetos se crean en una función o método no existen en otros ámbitos - otras funciones o métodos

Hay tres tipos principales de ámbitos en Python:

Ámbito global (Global scope): Las variables definidas fuera de cualquier función o bloque de código tienen un ámbito global. Estas variables pueden ser accedidas y modificadas desde cualquier lugar dentro del programa.

·

|  |  |
| --- | --- |
| ·Código Python   |  | | --- | | x = 10 # Variable global  def mi\_funcion():  print(x) # Podemos acceder a x desde esta función  mi\_funcion() |   · |

·

Ámbito de función (Function scope): Las variables definidas dentro de una función solo son visibles y accesibles dentro de esa función. No pueden ser accedidas desde fuera de la función.

·

|  |  |
| --- | --- |
| ·Código Python   |  | | --- | | def mi\_funcion():  y = 5 # Variable con ámbito de función, solo visible dentro de la función  print(y)  mi\_funcion()  # print(y) # Esto daría un error, ya que y no está definida fuera de la función |   · |

·

Ámbito de bloque (Block scope): A partir de Python 3.6, las variables definidas dentro de bloques como bucles for, while, condicionales if, etc., tienen un ámbito local limitado al bloque donde se definen.

·

|  |  |
| --- | --- |
| ·Código Python   |  | | --- | | for i in range(5):  z = i # Variable con ámbito de bloque, solo visible dentro del bucle  print(z)  # print(z) # Esto daría un error, ya que z no está definida fuera del bucle |   · |

·

Es importante entender el ámbito de las variables para evitar errores y comportamientos inesperados en el código. Además, para utilizar variables globales dentro de una función, se puede utilizar la palabra clave global para indicar que se quiere trabajar con la variable global en lugar de una local con el mismo nombre.

Definiendo nombres en el ámbito global con la expresión global.

locals()

·

|  |  |  |
| --- | --- | --- |
| Locals() devuelve un diccionario que representa el ámbito de variables locales (variables definidas en el ámbito actual, incluyendo variables locales, argumentos de función y variables temporales.) en el punto actual de ejecución.  ·Código Python   |  | | --- | | def mi\_funcion(a, b):  x = 10  y = 20  local\_vars = locals()  print(local\_vars)  mi\_funcion(5, 6) |   ·Salida esperada por consola   |  | | --- | | {'a': 5, 'b': 6, 'x': 10, 'y': 20} |   · |

·

Es importante tener en cuenta que locals() solo está disponible dentro de una función o método, ya que se refiere al ámbito local. Fuera de una función, se comportará como globals().

Globals()

·

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Globals() devuelve un diccionario que representa el espacio de nombres global actual en el contexto de ejecución. Este diccionario contiene todas las variables globales y sus valores en ese momento.  ·Código Python   |  | | --- | | x = 10  y = 20  def mi\_funcion():  global\_vars = globals()  print(global\_vars)  mi\_funcion() |   ·Salida esperada por consola   |  | | --- | | {'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, '\_\_loader\_\_': <...>, '\_\_spec\_\_': None, '\_\_annotations\_\_': {}, '\_\_builtins\_\_': <...>, 'x': 10, 'y': 20, 'mi\_funcion': <function mi\_funcion at 0x...>} |   ·   |  | | --- | | objeto\_1 = "Hola"  objeto\_2 = "Mundo IT"  namespace = globals()  print(f"original en {objeto\_1=}")  print(f"original en {objeto\_2=}")  print("\*"\*50)  #------------------------------------------------------------------------  # Modificando una variable global  namespace["objeto\_1"] = 20  namespace["objeto\_2"] = 3.14159  print(f"cambio en {objeto\_1=}")  print(f"cambio en {objeto\_2=}") |   ·Salida esperada por consola   |  | | --- | | original en objeto\_1='Hola'  original en objeto\_2='Mundo IT'  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  cambio en objeto\_1=20  cambio en objeto\_2=3.14159 |   · |

·

Es posible que una función pueda ligar un objeto al espacio de nombres del ámbito global mediante el uso de la expresión global con la siguiente sintaxis.

**global** objeto # variable

objeto = valor

|  |
| --- |
| Por flujo de información, las funciones deben existir 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.

**Resumen:** Cuando se invoca a una función y se hace una referencia a un nombre, el intérprete primeramente busca una coincidencia dentro del ámbito local y posteriormente en el ámbito global. En caso de no encontrarla, se generará un error de tipo NameError.

Arquitectura de una función.

Solo dos palabras reservadas. **def** crea las funciones y **return** devuelve al lugar que fue llamado el/los resultado(s) de la función o None por default

|  |
| --- |
| **def** mi\_funcion (parametro1, parametro2, ...):  # Código de la función  **return** # Opcionalmente, la función puede retornar un valor  # invocación  mi\_funcion (parametro1, parametro2, ...) |

·

Parámetros y argumentos

|  |
| --- |
| Definiciones |
| Es posible ingresar datos al ser invocadas a estos datos se les denomina argumentos y son ligados a nombres, los cuales se conocen como parámetros.  El número de argumentos ingresados debe corresponder al número de parámetros que se definen.  En caso de que no se ingresen los argumentos necesarios, se generará un error de tipo \_TypeError\_. |

·

.

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | 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

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | 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 |   · |

·

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.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | 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  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

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | 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:

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) |

Argumentos por nombre del parámetro (palabra clave):

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) |

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.

Parámetros con argumentos por defecto u omisión.

La sintaxis básica de argumentos por defecto en Python es la siguiente:

|  |
| --- |
| **def** mi\_funcion (parametro1=99, parametro2=66):  # parametro1 tendrá el valor de 99  # parametro2 tendrá el valor de 66  return  mi\_funcion () |

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:

* posicionales
* \* arbitrarios
* \*\* kwords -clave valor
* por omisión

Recursividad:

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

Cada vez que una función se invoca a si misma, Python crea un nuevo objeto de tipo function con las mismas características que la función original, pero con un ámbito totalmente nuevo y de nivel inferior a la función original.

,

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) |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def factorial\_recursivo(valor):  if valor == 1:  return 1  else:  return valor \* factorial\_recursivo(valor-1)  salida = 10#< -----------------modificar  regreso = factorial\_recursivo(salida) |   ·Salida esperada por consola   |  | | --- | | el factorial de 10 es 3628800 |   · |

·

|  |  |  |  |
| --- | --- | --- | --- |
| ·Código Python   |  | | --- | | 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() |   ·Salida esperada por consola   |  | | --- | | 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 |   ·Salida esperada por consola   |  | | --- | | Ingrese su password (1234) :1234#<---------------------------ingreso de la altura del usuario  Ingresaste!!!!!! |   · |

·

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def fibonacci(n):  if n <= 0:  return []  elif n == 1:  return [0]  elif n == 2:  return [0, 1]  else:  fib = fibonacci(n - 1)  fib.append(fib[-1] + fib[-2])  return fib  n = 10  resultado = fibonacci(n)  print(f"Los primeros {n} números de la serie de Fibonacci son: {resultado}") |   ·Salida esperada por consola   |  | | --- | | Los primeros 10 números de la serie de Fibonacci son: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34] |   · |

·

**Anotaciones en funciones** (function annotation)

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.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | 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} ") |   ·Salida esperada por consola   |  | | --- | | el fibonacci de 10 es 55 |   · |

·

Manejo de documentación de funciones - docstrings

|  |
| --- |
| 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 ')=}") |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | 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 desde funciones:

Todas las funciones regresan un valor al finalizar su ejecución al cual se le puede asignar un nombre si se desea conservarlo. Por defecto, el valor que regresan es **None**, el cual a diferencia de otros valores no es desplegado por el intérprete.

La expresión **return** se utiliza para regresar un objeto específico a su ámbito superior y acto seguido dar por terminada la ejecución de la función (hay excepciones – ver finnaly).

Pueden incluirse varias expresiones **return** en una función, pero sólo se ejecutará la primera que se encuentre.

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=}") |

Funciones anidadas.

Python permite definir funciones dentro de otras funciones.

En Python, es posible definir funciones anidadas, también conocidas como funciones internas o funciones dentro de funciones. Una función anidada se define dentro del cuerpo de otra función y puede acceder a las variables locales de la función externa. Esto puede ser útil para encapsular la lógica relacionada en un solo lugar y para limitar el alcance de ciertas variables.

·

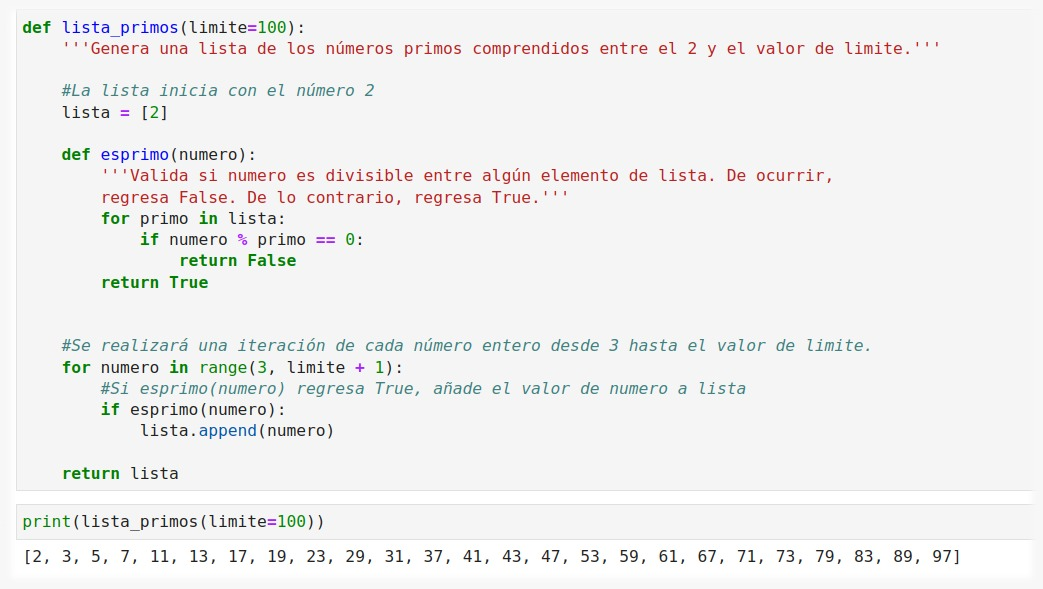
|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def funcion\_externa(x):  def funcion\_anidada(y):  return y \* 2  resultado = funcion\_anidada(x)  return resultado  valor = 5  resultado\_final = funcion\_externa(valor)  print(f"{resultado\_final=}") # Esto imprimirá 10 |   ·Salida esperada por consola   |  | | --- | | resultado\_final=10 |   · |

·

En este ejemplo, funcion\_anidada es una función anidada dentro de funcion\_externa. La función anidada toma un argumento y y lo multiplica por 2. La función externa funcion\_externa llama a la función anidada con un valor x, y luego devuelve el resultado.

Ten en cuenta que la función anidada solo es visible y accesible dentro del ámbito de la función externa. No se puede acceder a la función anidada desde fuera de la función externa.

Las funciones anidadas pueden ser útiles para encapsular ciertas operaciones dentro de una función más grande, lo que puede mejorar la legibilidad del código y mantener una estructura lógica. Sin embargo, en muchos casos, también es posible lograr el mismo resultado utilizando funciones independientes definidas fuera de la función principal.

En el ejemplo anterior se definió a la función esprimo() dentro de la función listaprimos(). Como se puede observar, el nombre lista está en el espacio de nombres de listaprimos(), pero al estar en un entorno superior al ámbito de esprimo(), éste puede acceder a lista.

Funciones lambda:

Las funciones lambda, también conocidas como funciones anónimas, son funciones pequeñas y de una sola línea que se definen sin un nombre utilizando la palabra clave lambda.

|  |
| --- |
| En realidad si bien la comunidad recomienda no usar nombres la mayoría de los ejemplos son con nombre |

Son útiles cuando necesitas una función simple para operaciones simples y temporales de una linea sin tener que definirla formalmente utilizando def.

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

|  |
| --- |
| lambda argumentos: expresion  argumentos:  Si los hay son los argumentos de la función separados por comas.  expresión:  Es una expresión que define el cuerpo de la función lambda.  Esta expresión se puede evalúar.  Se devuelve como resultado de la función. |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | suma = lambda a, b: a + b  print(f"{suma(3,6)=}") |   ·Salida esperada por consola   |  | | --- | | suma(3,6)=9 |   · |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | salida = lambda a, b: a + b if a>b else a - b  print(f"{salida(3,6)=}")  print(f"{salida(6,3)=}") |   ·Salida esperada por consola   |  | | --- | | salida(3,6)=-3  salida(6,3)=9 |   · |

·

Funciones de orden superior.

En Python todos son objetos, las funciones también.

Las funciones de orden superior en Python son aquellas funciones que pueden aceptar otras funciones como argumentos de entrada, manipularlas de manera flexible dentro de otras funciones y al final devolver funciones como resultado.

|  |
| --- |
| En Python, las funciones de orden superior se benefician del hecho de que las funciones son ciudadanos de primera clase, lo que significa que se pueden tratar como cualquier otro objeto, como enteros, cadenas o listas. Esto permite realizar operaciones avanzadas con funciones, como pasarlas como argumentos a otras funciones, asignarlas a variables, almacenarlas en estructuras de datos, retornarlas como resultado de una función, etc. |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def mi\_funcion(operacion):  if operacion == "suma":  def suma(a, b):  return a + b  return suma  elif operacion == "resta":  def resta(a, b):  return a - b  return resta  #--------------------------------------------------------  operacion = mi\_funcion("suma")  resultado = operacion(3, 4)  print(f"suma=",resultado) # Salida: 7  operacion = mi\_funcion("resta")  resultado = operacion(8, 4)  print(f"resta=",resultado) # Salida: 4  resultado = mi\_funcion("suma")(8, 2)  print(f"suma=",resultado) # Salida: 6 |   ·Salida esperada por consola   |  | | --- | | suma= 7  resta= 4  suma= 10 |   · |

·

map

La función map() en Python es una función de orden superior que se utiliza para aplicar una función original dada a cada elemento de un iterable (como una lista, tupla o set) y devuelve un objeto map que contiene los resultados. Es muy útil cuando necesitas aplicar una función a múltiples elementos de una colección iterable de manera eficiente y elegante de procesar datos en Python.

La sintaxis básica de la función map() es la siguiente:

|  |
| --- |
| map(función, iterable) |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def cuadrado(cada\_numero):  resultado = cada\_numero \*\* 2  return resultado  #--------------------------------------------------------  numeros = [1, 2, 3, 4, 5]  regreso = map(cuadrado, numeros)  print (f"{regreso=}")  print (f"{type(regreso)=}")  print (f"{list(regreso)=}") |   ·Salida esperada por consola   |  | | --- | | regreso=<map object at 0x00000aaaaaaaaaaa>  type(regreso)=<class 'map'>  list(regreso)=[1, 4, 9, 16, 25] |   · |

·

Se define la función estándar con el nombre cuadrado(parámetro) con un return parámetro al cuadrado (\*\*2)

Map aplica esta función a cada elemento de la lista números. Los regresos se coleccionan en un objeto map. Para ver su contenido hacemos un list(objeto map) o un for de cada dato map

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | palabras = ["Python", "es", "genial"]  # len como función original  longitudes = map(len, palabras)  print (f"{longitudes=}")  print (f"{type(longitudes)=}")  for palab, longi in zip (palabras,longitudes):  print (f"palabra:'{palab}' longitud-len= {longi}") |   ·Salida esperada por consola   |  | | --- | | longitudes=<map object at 0x00000bbbbbbbbbbb>  type(longitudes)=<class 'map'>  palabra:'Python' longitud-len= 6  palabra:'es' longitud-len= 2  palabra:'genial' longitud-len= 6  list(regreso)=[1, 4, 9, 16, 25] |   · |

·

Len es una función built-in incorporada para calcular la longitud de un string.

Map() se aplica a la función len() y se genera un iterable con las longitudes correspondientes.

Es importante tener en cuenta que map() devuelve un objeto map que es un iterable. Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.

filter

filter() es una función de orden superior que se utiliza para filtrar elementos de un iterable (como una lista, tupla o set) según una función original de filtrado dada.

Es importante tener en cuenta que filter() devuelve un objeto filter que es un iterable. Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.

La sintaxis básica de la función filter() es la siguiente:

|  |
| --- |
| filter(función, iterable) |

En este caso la función debe ser un filtro que solo regrese un True o un False.

Filter aplica a cada elemento del iterable de la colección y True se incluirá en el objeto filter mientras que los False serán excluido.

iterable: Es el objeto iterable (como una lista, tupla, conjunto) del que se desean filtrar los elementos.

La función filter() es útil cuando necesitas seleccionar elementos específicos de un iterable según una condición. Proporciona una forma concisa y eficiente de filtrar datos en Python.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def es\_par(numero):# original  return numero % 2 == 0  #--------------------------------------------------------  numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  solo\_pares = filter(es\_par, numeros)  print (f"{solo\_pares=}")  print (f"{type(solo\_pares)=}")  for cada\_numero in (solo\_pares):  print (f"\t\tnúmero {cada\_numero=}") |   ·Salida esperada por consola   |  | | --- | | solo\_pares=<filter object at 0x00000ccccccccccc>  type(solo\_pares)=<class 'filter'>  número cada\_numero=2  número cada\_numero=4  número cada\_numero=6  número cada\_numero=8  número cada\_numero=10 |   · |

·

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | cuadrado = lambda x: x \* x  print(cuadrado(5))  pares = filter(lambda x: x % 2 == 0, [1, 2, 3, 4, 5, 6])  pares\_lista = list(pares)  print(pares\_lista) |   ·Salida esperada por consola   |  | | --- | | 25  [2, 4, 6] |   · |

·

reduce

|  |
| --- |
| Es necesario recordar que reduce necesita ser importado de la librería functools |

reduce() es una función de orden superior que se utiliza para aplicar una función acumulativa original a los elementos de un iterable y reducirlos a un solo valor.

Esta función se encuentra en el módulo functools y requiere ser importada antes de su uso.

La sintaxis básica de la función reduce() es la siguiente:

|  |
| --- |
| from functools import reduce  reduce(función, iterable)  reduce(función, iterable, [initializer]) |

En este caso la función debe ser un filtro que solo regrese un True o un False.

Al ser una función acumulativa que se aplicará a los elementos del iterable. Esta función original debe tomar dos argumentos y devolver el resultado de la operación acumulativa.

iterable: Es el objeto iterable (como una lista, tupla, conjunto) que se desea reducir a un solo valor.

initializer (opcional): Es un valor inicial que se utiliza como el primer argumento en la primera llamada a la función acumulativa. Si no se proporciona, el primer elemento del iterable se utiliza como valor inicial.

Es importante tener en cuenta que reduce() requiere al menos dos elementos en el iterable. Si el iterable está vacío y no se proporciona un valor inicial (initializer), se generará un error.

La función reduce() es útil cuando necesitas realizar una operación acumulativa en los elementos de un iterable y reducirlos a un solo valor. Algunos ejemplos comunes de uso incluyen la suma de todos los elementos, el cálculo del producto de todos los elementos, la concatenación de cadenas, entre otros.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | from functools import reduce  def sumar(acumulador, valor\_nuevo ):  print (f"{acumulador=} {valor\_nuevo=}")  return valor\_nuevo + acumulador  #--------------------------------------------------------  numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  resultado = reduce(sumar, numeros)  print (f"{resultado=}")  print (f"{type(resultado)=}") |   ·Salida esperada por consola   |  | | --- | | acumulador=1 valor\_nuevo=2  acumulador=3 valor\_nuevo=3  acumulador=6 valor\_nuevo=4  acumulador=10 valor\_nuevo=5  acumulador=15 valor\_nuevo=6  acumulador=21 valor\_nuevo=7  acumulador=28 valor\_nuevo=8  acumulador=36 valor\_nuevo=9  acumulador=45 valor\_nuevo=10  resultado=55  type(resultado)=<class 'int'> |   · |

·

En este ejemplo, se define la función sumar(acumulador, valor\_nuevo) que toma dos argumentos y devuelve su suma return valor\_nuevo + acumulador.

Esto se itera para cada elemento de la colección y acumula sucesivamente los elementos del return

Pueden comparar con print (sum( numeros))

Funciones lambda + funciones de orden superior.

|  |
| --- |
| Cuadro sinóptico lambda |
| lambda parámetro : expresión  Funciones anónimas que en Python pueden llevar nombre :(   * + lambda + map   + lambda + filter   + lambda + reduce (import functools) |
| **Map, Filter y Reduce:** Estas son funciones incorporadas que son fundamentales en programación funcional.  **map(función, secuencia):** Aplica una función a cada elemento de una secuencia y devuelve un iterable con los resultados.  **filter(función, secuencia):** Filtra una secuencia usando una función de prueba y devuelve los elementos que pasan la prueba.  **reduce(función, secuencia):** Aplica una función a una secuencia de manera acumulativa para reducir la secuencia a un solo valor.  **Comprensiones de listas y generadores:** Estas son formas concisas de crear listas y generadores usando expresiones compactas.  cuadrados = [x \*\* 2 for x in range(10)]  **Inmutabilidad:** En programación funcional, los datos tienden a ser inmutables, lo que significa que no cambian después de ser creados. Las operaciones crean nuevos objetos en lugar de modificar los existentes.  **Recursividad:** Es una técnica fundamental en programación funcional. Las funciones recursivas se llaman a sí mismas para resolver problemas de manera iterativa.  **Clausuras y ámbitos léxicos:** Python permite crear clausuras, donde una función interna puede acceder y "capturar" variables locales de su función contenedora.  **Decoradores:** Los decoradores son una forma de modificar el comportamiento de una función o método en Python. Son ampliamente utilizados en programación funcional para agregar funcionalidades adicionales a las funciones.  **Programación funcional en bibliotecas:** Python tiene bibliotecas como functools que proporcionan herramientas para trabajar con funciones de manera funcional, y bibliotecas como itertools que ofrecen utilidades para manejar iteradores de manera funcional.  La programación funcional puede mejorar la legibilidad y mantenibilidad del código, fomentar la modularidad y la reutilización, y facilitar la concurrencia y el paralelismo. Aunque Python no es un lenguaje puramente funcional como Haskell, su soporte para programación funcional permite combinar este enfoque con otros paradigmas según sea necesario. |

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | string = "Python es el mejor lenguaje de programación"  palabras = string.split()  print (f"{palabras=}")  longitud\_mayor\_a\_5 = filter(lambda palabra: len(palabra) >= 5, palabras)  print(f"{list(longitud\_mayor\_a\_5)=}") |   ·Salida esperada por consola   |  | | --- | | palabras=['Python', 'es', 'el', 'mejor', 'lenguaje', 'de', 'programación']  list(longitud\_mayor\_a\_5)=['Python', 'mejor', 'lenguaje', 'programación'] |   · |

·

En este ejemplo, se utiliza una función lambda para verificar si la longitud de una cadena es mayor o igual a 5. filter() se aplica a la función lambda y a cada elemento de la lista de palabras, y se generan objeto tipo filter que es un iterable con las palabras con longitud >= a 5.

Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]  resultado = filter(lambda x: x % 2 == 0, todos)  print(f" la salida de filter es {resultado}")  print(f" type {type(resultado)}")  print(f" la lista de la salida de filter es {list(resultado)}") |   ·Salida esperada por consola   |  | | --- | | la salida de filter es <filter object at 0x000001E57FA8F790>  type <class 'filter'>  la lista de la salida de filter es [2, 4, 6, 8, 0] |   · |

·

La función filter() es útil cuando necesitas seleccionar elementos específicos de un iterable según una condición de filtrado. Puedes utilizar una función definida por el usuario o una función lambda para especificar la condición de filtrado. Esto te permite realizar operaciones más complejas de selección y filtrado en tus datos.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]  pares = [2, 4, 6, 8, 10]  resultado = filter(lambda xy: xy[0] in pares and xy[1] in pares, zip(todos, pares))  print(f" la salida de filter es {resultado}")  print(f" type {type(resultado)}")  print(f" la lista de la salida de filter es {list(resultado)}") |   ·Salida esperada por consola   |  | | --- | | la salida de filter es <filter object at 0x000001A994805720>  type <class 'filter'>  la lista de la salida de filter es [(2, 4), (4, 8)] |   · |

·

Una función lambda para verificar si los elementos de números y pares están presentes en la lista pares. filter() se aplica a la función lambda y a los iterables numeros y pares, y se generan los elementos que cumplen con la condición.

·

|  |
| --- |
| Cuadro sinóptico de Ámbitos en Python |
| Ámbito Local (Local Scope):  El ámbito local se refiere al ámbito dentro de una función o método.  Las variables definidas dentro de una función solo son accesibles dentro de esa función.  Si se intenta acceder a una variable local fuera de su función, se producirá un error. |
| Ámbito Encerrado (Enclosing Scope)  El ámbito encerrado se refiere al ámbito que se encuentra alrededor de una función anidada.  Ocurre cuando hay una función definida dentro de otra función.  Las variables definidas en el ámbito encerrado son accesibles tanto dentro de la función anidada como en la función externa. |
| Ámbito Global (Global Scope):  El ámbito global se refiere al ámbito fuera de cualquier función o clase.  Las variables definidas en el ámbito global son accesibles desde cualquier parte del código, incluyendo dentro de funciones y clases. |
| Ámbito de Nombre Integrado (Built-in Scope):  El ámbito de nombre integrado se refiere a las funciones y variables predefinidas que están disponibles en Python sin necesidad de importar ningún módulo.  Incluye funciones y objetos integrados como print(), len(), range(), etc.  Estas funciones y variables están disponibles en cualquier parte del código.  Es importante tener en cuenta que hay una jerarquía en la resolución de nombres en Python, lo que significa que Python busca una variable en el ámbito local primero, luego en el ámbito encerrado, seguido del ámbito global y finalmente en el ámbito de nombre integrado. Si no se encuentra la variable en ningún ámbito, se genera un error de "NameError". |

En este ejemplo, la variable x tiene un ámbito global y es accesible en todas partes del código. La variable y tiene un ámbito local dentro de la función func(), mientras que la variable z tiene un ámbito encerrado dentro de la función nested\_func(). Las variables x, y y z son accesibles dentro de sus respectivos ámbitos.

·

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | # Ámbito global  x = 10  y = "Sin Datos root"  z = "Sin Datos root"  def func():  # Ámbito local  y = 20  def nested\_func():  # Ámbito encerrado  z = 30  print(f"dentro de función anidada: {x=} {y=} {z=}") # Accede a las variables x, y y z  nested\_func()  print(f"dentro de función: {x=} {y=} {z=}") # Accede a las variables x e y  func()  print(f"root: {x=} {y=} {z=}") # Accede a la variable x |   ·Salida esperada por consola   |  | | --- | | dentro de función anidada: x=10 y=20 z=30  dentro de función: x=10 y=20 z='Sin Datos root'  root: x=10 y='Sin Datos root' z='Sin Datos root' |   · |

·

Funciones como argumento de entrada a otra función:

|  |  |  |
| --- | --- | --- |
| ·Código Python   |  | | --- | | def operacion(func\_parametro, numeros):  #------------------------------------------------------------------------  # opcion 1  # ~ resultados = []  # ~ for num in numeros:  # ~ resultados.append(func\_parametro(num))  #------------------------------------------------------------------------  # opcion 2  # ~ resultados = [func\_parametro(num) for num in numeros ]  #------------------------------------------------------------------------  # opcion 3  resultados = list(map(func\_parametro,numeros) )  #------------------------------------------------------------------------  return resultados  def cuadrado(num):  return num \*\* 2  def cubo(num):  return num \*\* 3  numeros = [1, 2, 3, 4, 5]  r\_cuadrado = operacion(cuadrado, numeros)  r\_cubo = operacion(cubo, numeros)  print(f"{r\_cuadrado=}")  print(f"{r\_cubo=}") |   ·Salida esperada por consola   |  | | --- | | r\_cuadrado=[1, 4, 9, 16, 25]  r\_cubo=[1, 8, 27, 64, 125 |   · |

·

|  |
| --- |
| Tabla de punteo de palabras reservadas  y funciones incorporadas |
| ·   |  |  |  |  | | --- | --- | --- | --- | | Palabras reservada Python 3.12 que ya deberías conoces | | | | | False  None  True  **A**  and  as  **B**  break  **C**  class  continue | **D**  def  del  **E**  elif  else  except  **F**  finally  for  from  **G**  global | **I**  if  import  in  is  **L**  lambda  **N**  nonlocal  not  **O**  or | **P**  [p](https://docs.python.org/es/3.12/library/functions.html" \l "pow)ass  **R**  raise  return  **T**  try  **W**  while  with |   · |

·

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ·   | Funciones Built-in Python 3.12 que ya deberías poder usarlas | | | | | --- | --- | --- | --- | | **A**  [abs()](https://docs.python.org/es/3.12/library/functions.html" \l "abs)  [all()](https://docs.python.org/es/3.12/library/functions.html" \l "all)  [any()](https://docs.python.org/es/3.12/library/functions.html" \l "any)  [ascii()](https://docs.python.org/es/3.12/library/functions.html" \l "ascii)  **B**  [bin()](https://docs.python.org/es/3.12/library/functions.html" \l "bin)  [bool()](https://docs.python.org/es/3.12/library/functions.html" \l "bool)  [bytearray()](https://docs.python.org/es/3.12/library/functions.html" \l "func-bytearray)  [bytes()](https://docs.python.org/es/3.12/library/functions.html" \l "func-bytes)  **C**  [chr()](https://docs.python.org/es/3.12/library/functions.html" \l "chr)  [complex()](https://docs.python.org/es/3.12/library/functions.html" \l "complex)  **D**  [dict()](https://docs.python.org/es/3.12/library/functions.html" \l "func-dict)  [dir()](https://docs.python.org/es/3.12/library/functions.html" \l "dir)  [divmod()](https://docs.python.org/es/3.12/library/functions.html" \l "divmod) | **E**  [enumerate()](https://docs.python.org/es/3.12/library/functions.html" \l "enumerate)  [eval()](https://docs.python.org/es/3.12/library/functions.html" \l "eval)  **F**  [filter()](https://docs.python.org/es/3.12/library/functions.html" \l "filter)  [float()](https://docs.python.org/es/3.12/library/functions.html" \l "float)  [frozenset()](https://docs.python.org/es/3.12/library/functions.html" \l "func-frozenset)  **G**  globals()  **H**  [help()](https://docs.python.org/es/3.12/library/functions.html" \l "help)  [hex()](https://docs.python.org/es/3.12/library/functions.html" \l "hex)  **I**  [id()](https://docs.python.org/es/3.12/library/functions.html" \l "id)  [input()](https://docs.python.org/es/3.12/library/functions.html" \l "input)  [int()](https://docs.python.org/es/3.12/library/functions.html" \l "int)  [isinstance()](https://docs.python.org/es/3.12/library/functions.html" \l "isinstance) | **L**  [len()](https://docs.python.org/es/3.12/library/functions.html" \l "len)  [list()](https://docs.python.org/es/3.12/library/functions.html" \l "func-list)  [locals()](https://docs.python.org/es/3.12/library/functions.html" \l "locals)  **M**  [map()](https://docs.python.org/es/3.12/library/functions.html" \l "map)  [max()](https://docs.python.org/es/3.12/library/functions.html" \l "max)  [min()](https://docs.python.org/es/3.12/library/functions.html" \l "min)  **O**  **[object](https://docs.python.org/es/3.12/library/functions.html" \l "object)**()  [oct()](https://docs.python.org/es/3.12/library/functions.html" \l "oct)  [open()](https://docs.python.org/es/3.12/library/functions.html" \l "open)  [ord()](https://docs.python.org/es/3.12/library/functions.html" \l "ord)  **P**  [pow()](https://docs.python.org/es/3.12/library/functions.html" \l "pow)  [print()](https://docs.python.org/es/3.12/library/functions.html" \l "print) | **R**  [range()](https://docs.python.org/es/3.12/library/functions.html" \l "func-range)  [reversed](https://docs.python.org/es/3.12/library/functions.html" \l "reversed)()  [round()](https://docs.python.org/es/3.12/library/functions.html" \l "round)  **S**  [set()](https://docs.python.org/es/3.12/library/functions.html" \l "func-set)  [sorted()](https://docs.python.org/es/3.12/library/functions.html" \l "sorted)  [str()](https://docs.python.org/es/3.12/library/functions.html" \l "func-str)  [sum()](https://docs.python.org/es/3.12/library/functions.html" \l "sum)  **T**  [tuple()](https://docs.python.org/es/3.12/library/functions.html" \l "func-tuple)  [type()](https://docs.python.org/es/3.12/library/functions.html" \l "type)  **Z**  [zip()](https://docs.python.org/es/3.12/library/functions.html" \l "zip) |   · |

·

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ·   |  |  |  |  | | --- | --- | --- | --- | | Funciones Built-in en Python 3.12  <https://docs.python.org/es/3.12/library/functions.html> | | | | | **A**  [abs()](https://docs.python.org/es/3.12/library/functions.html" \l "abs)  [aiter()](https://docs.python.org/es/3.12/library/functions.html" \l "aiter)  [all()](https://docs.python.org/es/3.12/library/functions.html" \l "all)  [any()](https://docs.python.org/es/3.12/library/functions.html" \l "any)  [anext()](https://docs.python.org/es/3.12/library/functions.html" \l "anext)  [ascii()](https://docs.python.org/es/3.12/library/functions.html" \l "ascii)  **B**  [bin()](https://docs.python.org/es/3.12/library/functions.html" \l "bin)  [bool()](https://docs.python.org/es/3.12/library/functions.html" \l "bool)  [breakpoint()](https://docs.python.org/es/3.12/library/functions.html" \l "breakpoint)  [bytearray()](https://docs.python.org/es/3.12/library/functions.html" \l "func-bytearray)  [bytes()](https://docs.python.org/es/3.12/library/functions.html" \l "func-bytes)  **C**  [callable()](https://docs.python.org/es/3.12/library/functions.html" \l "callable)  [chr()](https://docs.python.org/es/3.12/library/functions.html" \l "chr)  [classmethod()](https://docs.python.org/es/3.12/library/functions.html" \l "classmethod)  [compile()](https://docs.python.org/es/3.12/library/functions.html" \l "compile)  [complex()](https://docs.python.org/es/3.12/library/functions.html" \l "complex)  **D**  [delattr()](https://docs.python.org/es/3.12/library/functions.html" \l "delattr)  [dict()](https://docs.python.org/es/3.12/library/functions.html" \l "func-dict)  [dir()](https://docs.python.org/es/3.12/library/functions.html" \l "dir)  [divmod()](https://docs.python.org/es/3.12/library/functions.html" \l "divmod) | **E**  [enumerate()](https://docs.python.org/es/3.12/library/functions.html" \l "enumerate)  [eval()](https://docs.python.org/es/3.12/library/functions.html" \l "eval)  [exec()](https://docs.python.org/es/3.12/library/functions.html" \l "exec)  **F**  [filter()](https://docs.python.org/es/3.12/library/functions.html" \l "filter)  [float()](https://docs.python.org/es/3.12/library/functions.html" \l "float)  [format()](https://docs.python.org/es/3.12/library/functions.html" \l "format)  [frozenset()](https://docs.python.org/es/3.12/library/functions.html" \l "func-frozenset)  **G**  [getattr()](https://docs.python.org/es/3.12/library/functions.html" \l "getattr)  [globals()](https://docs.python.org/es/3.12/library/functions.html" \l "globals)  **H**  [hasattr()](https://docs.python.org/es/3.12/library/functions.html" \l "hasattr)  [hash()](https://docs.python.org/es/3.12/library/functions.html" \l "hash)  [help()](https://docs.python.org/es/3.12/library/functions.html" \l "help)  [hex()](https://docs.python.org/es/3.12/library/functions.html" \l "hex)  **I**  [id()](https://docs.python.org/es/3.12/library/functions.html" \l "id)  [input()](https://docs.python.org/es/3.12/library/functions.html" \l "input)  [int()](https://docs.python.org/es/3.12/library/functions.html" \l "int)  [isinstance()](https://docs.python.org/es/3.12/library/functions.html" \l "isinstance)  [issubclass()](https://docs.python.org/es/3.12/library/functions.html" \l "issubclass)  [iter()](https://docs.python.org/es/3.12/library/functions.html" \l "iter) | **L**  [len()](https://docs.python.org/es/3.12/library/functions.html" \l "len)  [list()](https://docs.python.org/es/3.12/library/functions.html" \l "func-list)  [locals()](https://docs.python.org/es/3.12/library/functions.html" \l "locals)  **M**  [map()](https://docs.python.org/es/3.12/library/functions.html" \l "map)  [max()](https://docs.python.org/es/3.12/library/functions.html" \l "max)  [memoryview()](https://docs.python.org/es/3.12/library/functions.html" \l "func-memoryview)  [min()](https://docs.python.org/es/3.12/library/functions.html" \l "min)  **N**  [next()](https://docs.python.org/es/3.12/library/functions.html" \l "next)  **O**  [object()](https://docs.python.org/es/3.12/library/functions.html" \l "object)  [oct()](https://docs.python.org/es/3.12/library/functions.html" \l "oct)  [open()](https://docs.python.org/es/3.12/library/functions.html" \l "open)  [ord()](https://docs.python.org/es/3.12/library/functions.html" \l "ord)  **P**  [pow()](https://docs.python.org/es/3.12/library/functions.html" \l "pow)  [print()](https://docs.python.org/es/3.12/library/functions.html" \l "print)  [property()](https://docs.python.org/es/3.12/library/functions.html" \l "property) | **R**  [range()](https://docs.python.org/es/3.12/library/functions.html" \l "func-range)  [repr()](https://docs.python.org/es/3.12/library/functions.html" \l "repr)  [reversed()](https://docs.python.org/es/3.12/library/functions.html" \l "reversed)  [round()](https://docs.python.org/es/3.12/library/functions.html" \l "round)  **S**  [set()](https://docs.python.org/es/3.12/library/functions.html" \l "func-set)  [setattr()](https://docs.python.org/es/3.12/library/functions.html" \l "setattr)  [slice()](https://docs.python.org/es/3.12/library/functions.html" \l "slice)  [sorted()](https://docs.python.org/es/3.12/library/functions.html" \l "sorted)  [staticmethod()](https://docs.python.org/es/3.12/library/functions.html" \l "staticmethod)  [str()](https://docs.python.org/es/3.12/library/functions.html" \l "func-str)  [sum()](https://docs.python.org/es/3.12/library/functions.html" \l "sum)  [super()](https://docs.python.org/es/3.12/library/functions.html" \l "super)  **T**  [tuple()](https://docs.python.org/es/3.12/library/functions.html" \l "func-tuple)  [type()](https://docs.python.org/es/3.12/library/functions.html" \l "type)  **V**  [vars()](https://docs.python.org/es/3.12/library/functions.html" \l "vars)  **Z**  [zip()](https://docs.python.org/es/3.12/library/functions.html" \l "zip)  **\_**  [\_\_import\_\_()](https://docs.python.org/es/3.12/library/functions.html" \l "import__) |   · |