#help(''.lstrip)

#help ayuda y permite ver que hace una función

#dir(objeto) te indica todo lo que puedes hacer con cierto objeto

#title() capitaliza las iniciales de un texto

#'hola mundo'.title()

#imprime Hola Mundo

Os.path.isfile(ruta)

Esta función verifica si un objeto dentro un directorio/carpeta es un archivo

Os.path.isdir(ruta)  
Esta función verifica si un objeto es un directorio

AMBITO DE IDENTIFICADORES (SCOPE)

Un identificador es cualquier cosa que el programador pueda nombrar en el programa: variables, nombres de funciones, de clases, etc.  
El ámbito o scope se refiere al área del programa donde se puede tener acceso a un identificador, por ejemplo si defines una variable dentro de una función, no puedes tener acceso a dicha variable fuera de dicha función.

Los ámbitos se anidan, esto es, están dentro de otros: si un identificador no se encuentra en el ámbito actual, se busca en el siguiente nivel de ámbito

Si dos identificadores se llaman igual pero se encuentran en ámbitos diferentes, entonces se toma el identificador más cercano al ámbito activo

Si se definen dos identificadores en el mismo nivel de ámbito se hará una sobreescritura quedándose solo la ultima definición. En otros lenguajes como java podría generarse un error similar

TIPOS DE AMBITOS

Global:

Es el ámbito más general en Python, se puede identificar fácilmente pues no hay indentacion. Este ámbito se crea cuando inicia el programa y muere cuando termina el programa. Es una mala practica bien conocida el usar variables globales, pues son la fuente de errores por efectos colaterales. En este ámbito solo deberían definirse funciones, clases o constantes, no es recomendado definir variables.

Función/método:  
Los que se definen dentro del cuerpo de una función/método. Se crea un ámbito de este tipo por cada invocación a función/método, y el ámbito muere cuando este termina. En lenguajes como Python, se pueden definir funciones dentro de funciones. Las funciones declaradas dentro de funciones están en el ámbito de esa función.

Def externa()

.Def interna(): #funcion dentro de función  
. return 1  
.return interna() #se hace la referencia  
.

.

.

print(externa()) #sin errores

Print(interna()) #error, no está en el ámbito global

Clase/Objeto:

Se crea un ámbito de este tipo cada que se crea un objeto. El ámbito muere cuando el objeto pierde todas sus referencias, visualmente son los identificadores definidos dentro del cuerpo de una clase. Python realmente no tiene establecido este ámbito a diferencia de lenguajes como java, por eso se requiere el uso explicito de self para hacer referencia al ámbito del objeto.

Class Prueba():

Def\_\_init\_\_(self, propiedad):

self.propiedad = propiedad #ambitos diferentes

Def método(self):

Print(propiedad) #error, no existe  
 print(self.propiedad) #no hay error

Bloque:  
Es un tipo de ámbito que se crea cuando no hay sentencias de bloque, como las sentencias de control, y muere cuando termina el bloque, Python no tiene este ámbito pero hay lenguajes como java que si lo tienen.

If true:

Variable = 6 #definido en global

Print(variable) #no hay error

Recolección de basura:

Cuando el ámbito de un identificador desaparece, también desaparecen sus identificadores asociados. Considérese que en el caso de las variables, estas solo son referencias a posiciones en memoria y puede haber varias referencias a la misma memoria. Cuando un ámbito termina, es posible que desaparezcan referencias a datos en memoria. Si un dato en memoria no tiene referencias vivas (ósea ámbitos activos) entonces es propensa a ser reciclada. A este proceso de reciclado se le conoce como recolección de basura, el cual es un proceso especial del interpreta que se encarga de reciclar memoria que ya no tiene referencias vivas y que por lo tanto no puede ser recuperada. Muchos lenguajes tienen estos procesos, como java, Python, JavaScript, c, etc. Mientras que en otros lenguajes como c y c++, la liberación de memoria es un proceso manual que le corresponde al programador, lo cual tiene ventajas y desventajas

L1= [1,2,3]

L2=l1 #misma memoria  
l2[0] = 99  
print(l1[0]) #imprime 99

Paso de parámetros a funciones:  
Se refiere a los parámetros (o argumentos) que se pasan al mandar a llamar a una función. Dependiendo del tipo de dato, es posible que los parámetros se manden de forma diferente. Entender esas diferencias es fundamental en programación para prevenir y diagnosticar errores, pues lo primero es entender que las variables son referencias a direcciones en memoria, por si mismas no guardan valores, solo señalan donde se encuentra un dato. Hay dos formas de pasar valores:

.

. Paso por valor:  
. La forma en la que se pasan tipos simples (primitivos). Los que se pasan de esta forma se pueden entender como copias del valor original, de esta forma, si modificas el valor (la referencia realmente) no afecta a otras partes del código. Aunque realmente en lenguajes como Python que no tienen primitivos (pues los tipos simples y todos los tipos son objetos) se siguen pasando referencias.

Def fun(var):

. Var = 1 #la referencia se va a otro lado  
. return var

Var = 33 #este identificador está en otro ámbito  
fun(var)

Print(var) #no se afectó var e imprime 33

.

.

.

.

. Paso por referencia

. Es la forma en la que se pasan los tipos estructurados. Se puede entender mas directamente como el paso de referencias a memoria y no como copias de valores. Hay que tener cuidado, por que por lo general la mayoría de errores son causados por este paso

Def(fun(lista): #paso por referencia

. Lista[0] = 66 #se modifica memoria  
. return 1  
  
l = [1,2,3,4]  
fun(l)  
print(l) #imprime [66, 2, 3, 4]

Efectos Colaterales:  
Dado que al pasar por referencia se puede manipular memoria, es posible que al hacer cambios en una función se afecte a otras partes del código. A lo anterior se le conoce como efecto colateral, esto es tras ejecutar la función se produjeron resultados que posiblemente no se esperaban a priori, son una fuente común de bugs. Una manera de solucionar el problema es haciendo copias, si se quiere estar seguro de que no hay efectos colaterales, se pueden usar también estructuras no mutables como las tuplas. Los efectos colaterales también tienen que ver con crear archivos, modificar configuraciones del sistema o hacer cualquier cosa que no sea obvia y que cambia el estado del sistema de formas impredecibles y silenciosas

Def sumatoria(lista):

Res = 0  
 while lista:  
 res += lista.pop() #se va destruyendo la lista, efecto colateral  
 return res  
  
l = [1, 2, 3, 4]  
print(sumatoria(l)) #aparentemente todo está bien, imprime 10  
print(sumatoria(l)) #no era lo que se esperaba, imprime 0

CORREGIDO

Def sumatoria(lista):  
 lista = lista[:] #se hace una copia de la lista

Res = 0  
 while lista:  
 res += lista.pop() #se va destruyendo la copia  
 return res  
  
l = [1, 2, 3, 4]  
print(sumatoria(l)) #aparentemente todo está bien, imprime 10  
print(sumatoria(l)) #todo bien, imprime 10

Tipos de parámetros en las funciones:  
Posicionales obligatorios:  
Son el tipo de parámetro que se han estado usando, dependen de un orden por lo que se les llama posicionales. En Python es obligatoria pasar todos los parámetros posicionales al invocar la función

def fun(pos1, pos2, pos3):

Return pos1+pos2+pos3  
  
fun(1,2,3) #el mapeo es pos1=1, pos2=2, pos3=3, si se pasan de mas o de menos da error, porque solo soporta para 3

Opcionales:  
Como su nombre lo indica son opcionales, esto es, se pueden pasar o no al invocar la función, son posicionales en el sentido de que se corresponden con la posición de acuerdo a como se invoca la función, al declarar la función es obligatorio establecer el valor por defecto del parámetro, de hecho esta es la forma de distinguirlos de parámetros posicionales normales, al declarar la función, deben de ir después de los parámetros posicionales obligatorios o de lo contrario se confundirá el interprete

def fun(normal, opcional=0):  
 return normal + opcional  
  
print(fun(3)) #no se paso segundo parámetro, valor 0 por defecto, imprime 3  
print(fun(3, 4)) #si se pasó, se considera valido, imprime 7

otro ejemplo  
def fun(normal, opcional=0, op2=1):  
 return normal + opcional + op2  
  
print(fun(3)) #no se paso segundo parámetro, valor 0 por defecto, imprime 3  
print(fun(3, 4)) #si se pasó, se considera valido, imprime 7  
print(fun(3, op2=11) #pasa, imprime 14

Keyword(nombrados):  
No posicionales y se pueden pasar en el orden que sea. Se declaran igual que los parámetros opcionales, esto es, en Python, cualquier parámetro definido como opcional es también keyword (esto no es así en todos los lenguajes) Al igual que los opcionales, se deben definir después de los parámetros posicionales obligatorios. Al invocar la función, se puede pasar directamente la relación de valores del parámetro utilizando su nombre.

Def fun(normal, nombrado1=0, nombrado2=0):  
 return normal+nombrado1+nombrado2  
print(fun(1)) #aún son opcionales, imprime 1  
print(fun(1, nombrado2=11)) #puede pasar el que quiera, imprime 12  
print(fun(1, nombrado2=11, nombrado1=22)) #se puede pasar en diferente orden, imprime 34

Variable:  
Son un tipo especial de parámetro que representa un numero variable de parámetros, permiten que una función pueda recibir cualquier número de parámetros sin importar como fue definida, al definir la función, cualquier argumento que inicie con \* se considera variable. Solo puede haber uno de estos y debe ir al final de la lista de parámetros (o puede ser casi al final como se menciona más adelante). Todos los parámetros extra que pases se van a ir a una tupla. Es posible pasar una lista (u otra estructura secuencial) a una función que recibe parámetros variables mediante un proceso que se llama desempaquetar. Básicamente se agrega un \* al pasar la lista en la invocación de la función.

Def sumatoria(\*args):  
 res=o  
 for val in args:  
 res += val  
 return res  
print(sumatoria()) #imprime 0  
print(sumatoria(1)) #imprime 1  
print(sumatoria(1, 2)) #imprime 3  
print(sumatoria(1, 2, 3)) #imprime 6

Otro ejemplo:  
def fun(param1, op1=0 \*resto):  
 print(resto)  
fun(1) #al menos se debe pasar un parámetro, imprime ()  
fun(1,2) #imprime ()  
fun(1,2,3) #imprime (3,)

Fun(1,2,3,4) #imprime (3,4)

Otro ejemplo:

Def sumatoria(\*args):

Res = 0  
 for val in args:  
 res += val  
 return res  
lista = [1, 2, 3, 4]  
print(sumatoria(\*lista)) #se pasa como parámetros, imprime 10

Variables nombrados:

Un caso especial de parámetros variables son los parámetros variables nombrados. Es una idea similar, pero para pasar múltiples parámetros nombrados. Se utiliza \*\* al declarar el parámetro. Los parámetros nombrados extra se van a un diccionario, deben ir al final de la lista de parámetros, después de los parámetros variables. Al igual que los parámetros variables normales, se pueden desempaquetar valores, en este caso un diccionario, para desempaquetar se usa \*\* para invocar.

def fun(\*\*kargs):  
print(kargs)  
fun(nombre='pepe', apellido='pecas', edad=15) #imprime {'nombre': 'pepe', 'apellido': 'pecas', 'edad': 15}

otro ejemplo:  
def fun(\*\*kargs):  
print(kargs)  
diccionario = {'nombre': 'pepe', 'edad': 15, 'apellido': 'pecas'}  
fun(\*\*diccionario) #imprime {'nombre': 'pepe', 'edad': 15, 'apellido': 'pecas'}

Sobrecarga de funciones:  
Se refiere a poder definir varias veces la misma función pero con variación en los parámetros. En Python este concepto no existe. Al tenerse tipos dinámicos no hay necesidad en muchas ocasiones. Si necesitas que haya variaciones en el número de parámetros puedes usar parámetros opcionales o nombrados. Si se intenta, se hará una sobreescritura, esto es, se remplaza la función original por la última que se definió.

def sumar(val):  
return val + 1  
def sumar(val1, val2):  
return val1 + val2  
print(sumar(1)) # error, se esperaban dos parámetros

ejemplo que no tiene nada que ver pero vale la pena mencionar  
import datetime #permite ver la fecha  
f = datetime.now  
f.day  
f.month  
f.year  
f.hour

Documentación de funciones:  
La documentación es información que se provee para quienes quieren utilizar tu código, por ejemplo, la función help muestra la documentación de ayuda que el programador establecio  
help(función)  
ejemplo: help(print), help(‘’.split), help(‘’.capitalize)  
Se pueden documentar módulos, funciones, métodos y clases. La documentación la establece el programador con cadenas especial, usualmente es una cadena multilínea. Una buena documentación de funciones incluye: una descripción breve de lo que hace la función, una descripción breve de cada parámetro y una descripción breve de que devuelve.

def sumatoria(lista):

"""

Calcula la sumatoria de los elementos de lista.

lista: lista de números de entrada

returns: un número con la sumatoria

"""

res = 0

for elemento in lista:

res += elemento

return res

Anotaciones de tipos:  
Entre otras cosas son una forma de mejorar la documentación de un función, sirven para establecer el tipo de datos que se espera reciba y regrese la función, no establecen restricciones de tipo como en lenguajes como c, son simplemente un apoyo.

def sumatoria(lista: list) -> int:

"""

Calcula la sumatoria de los elementos de lista.

lista: list, lista de enteros

returns: int, sumatoria total

"""

res = 0

for elemento in lista:

res += elemento

return res

os.path.isfile y os.path.isdir revisa si es archivo o es ruta