**Intro to Python3 for CSC148 PART2 –** Version - October 2024

Posted at: <http://athena.ecs.csus.edu/~mitchell/csc148/Python3/Python3languageElements_f24_PART2.doc>x

by W Mitchell, Sacramento State U., College of Engineering and Computer Science, Computer Science Department

**Functions**

The **def** statement specifies a function name, followed by parameters in parentheses (*empty parentheses for* *parameter-less functions*.)

>>> def foo(\*, x): *< -- Colon terminates function header, and starts the scope of the (indented) function body*

*… “”” Print contents of passed argument x < -- CSC148-Mandatory (shop standard) docString*

*… “””*

... print (x) *<- - Function body is a block of statements that must be indented (the preamble docstring is also part of the body)*

...

>>> foo(x=123) *<- - A call to fcn foo passing type int argument 123; the call argument list does not have a first argument corresponding to \**

123

In the above print statement, if x was a list or other container instance, instead of a scalar, the container’s contents would be printed, even when the container object (such as a dictionary) is not a sequence.

A function is a “first class” object, as illustrated by:

>>> foo

<function foo at 0x0000012175226D90>

A function identifier (i.e., a function’s name) can be stored in a list, inspected, etc.

Example**:**

>>> def f(): < - - f is a dummy function (pass means no action – a pass statement specifies a do-nothing function body

pass

>>> lis =[1,'abc',f] *< - - The last element of lis is not a function call; it stands for the object that is the source code of f*

>>> lis[2]

<function f at 0x0000018E0B2BEB70> *< == As with everything in Python, a function definition, that is, its code, is an object*

**Argument and Parameter association**

*Terminology Note – even official Python documentation is inconsistent about the “parameters” of function vs a function’s “arguments”.*

*This class (148) will be CONSISTENT and use the proper terminology:*

*Given def f(\*,…, p, … )* ***“p” is a parameter of f****, and in the call f(…, p=a, …),* ***a is a call argument*** *corresponding to p.*

*\*\*\* The syntax in f’s signature and the call is explained next \*\*\**

The default (as in many languages) is positional parameters: the order/position of function parameters must be the same as the order/position of function arguments that appear in a call

The next section (i.e., Keyword parameters) describes how all homeworks are required to code function arguments & parameters.

**Keyword-only parameters**

As in many languages, the syntax: parameterName=value specifies a keyword parameter

**ALL Python (thus also simpy) functions in CSC148 MUST USE KEYWORD-ONLY PARAMETERS -**

“Must Use Keyword-Only Parameters” in this section’s title means

EVERY function parameter must be a keyword parameter (no positional parameters can be used – this is a “shop standard”, that is, a required way to code function calls and argument transmission; it is NOT required in Python per se.

The readability of keyword parameters is a great advantage. Moreover, the Python translator can enforce keyword-only parameters by specifying one \* character as the first item in a function’s signature.

Example:

"""

keywordOnlyParameters.py - Demo enforcing required use of keyword-only parameters

"""

def kwfcn(\*,length,width): # When the first parameter is a ‘\*’, mandatory keyword-only arguments are enforced by the translator

"""

Calculate & return the rectangle area for rectangle dimensions length & width

"""

return length\*width

# Required call syntax

print("Area is: ",kwfcn(length=13,width=8)) # All arguments are keyword (no arguments can be positional)

================================= RESTART: C:/Users/mitchell/148\_f22/keywordOnlyParameters.py ================================

Area is: 104

# An illegal call caused by failure to use keyword-only parameters convention during a function call

print("Area is: ",kwfcn(13,width=8)) # Demo automatic keyword-only enforcement; positional parameter “13” violates our shop standard

Traceback (most recent call last):

File "C:/Users/mitchell/148\_f22/keywordOnlyParameters.py", line 14, in <module>

print("Area is: ",kwfcn(13,width=8))

TypeError: kwfcn() takes 0 positional arguments but 1 positional argument (and 1 keyword-only argument) were given

**Returning values from a function call**

When a Python function returns

**a)** a scalar, that scalar value is returned

**b)** several values v1, …,vn using return v1,v2, … ,vn, the tuple (v1, …,vn) is returned

**c)** a container (such as a list/dict/tuple/set), that container is returned

**d)** has no return statement OR all return statements are simply RETURN, None is returned

Examples

Given the dummy function f, illustrated above,

>>> print("A call to f returns ", f())

Results of print call

A call to f returns None

def g(\*,x, y):

return x-y, x\*y

g # Verify that g is a defined function

<function g at 0x0000029ECB3F1090>

g(x=2,y=3)

(-1, 6) # A tuple of values is returned, and a specific element or tuple slice could also be returned/displayed, as needed

Example

print(g(x=2,y=3)[-1]) # Display the last returned tuple element

6

**Argument transmission mode**

Each parameter “p” corresponding to function call argument “ca” is a local variable during function execution. p is assigned the value of ca at start of function execution. The Python parameter transmission mechanism is referred to as

passing arguments by assignment.

(*This parameter passing mechanism is different from other languages that impleent: call by name, call by value,*

*call by reference, etc.)*

Review: Python’s semantics for [assignment statements](https://docs.python.org/3/reference/simple_stmts.html#assignment-statements) applies to a parameter during function execution. Recall the mechanics of assignment:

* If the assignment target is an identifier, or variable name, then this name is bound to the object. For example,

For assignment x = 2, x is the name/reference and 2 is the object

* Before the assignment, if the name was bound to a separate object, then that name is re-bound to the new object.

For example, if x’s value is currently 2,

and  x = 3 is executed, then the variable name x is re-bound to 3.

‘passing arguments by assignment’ – the Python argument transmission mode

Consider a function f(\*, …, p, …), its parameter p, and call f(…,p=a, …) *< -- call argument in the calling code is a*

1. When p is modified during f execution by an assignment, upon return from execution, a is unchanged
2. When p is modified during f execution in any way other than by assignment, a has p’s last value prior to return from function execution
3. In the special case that p is not modified during f execution, after return from f execution, a still has the value it had when f was called

Contrast/Compare rules a) and b) above to other languages – for example, Java uses call-by-value (thus a function g’s execution cannot modify the argument z passed to g (regardless of how the parameter p associated with z is manipulated during function execution).

However, Python’s effect a) above corresponds to a Java call for which an object value is passed, and Python’s effect b) above corresponds to a Java call for which a reference to an object is passed )

Script example Demo passing arguments by assignment

"""

passArgumentsByAssignment.py - Demo Python argument passing to functions

Version2 - Modify function parameters to (enforced) keyword-only parameters, Oct 2021

Version1 - WJM, CSUS, Sep 2020

"""

def f(\*,x):

"""

Demo passed arguments that are re-assigned in f or modified without assignment

"""

assert (isinstance(x,int)) or (isinstance(x,list)), "Argument must be int or list"

print("Start f execution; x is ", x," and its id is ",id(x))

if (isinstance(x,int)):

x = -107 # Modify x by assignment (assuming argument was not -107)

else:

x.append([0,0,0]) # Append a list to the parameter, thus modifying, but not assigning x

# x = [4,5,6] # Assign a new list to the parameter

print("Finishing f execution; x is ", x," and its id is ",id(x))

return

def g(\*,x):

""" Demo another list argument modification """

assert isinstance(x,list), "Argument must be a list"

print("Start g execution; x is ", x," and its id is ",id(x))

x.reverse() # Modify (not assign!) passed argument value

print("Finishing g execution; x is ", x," and its id is ",id(x))

return

y = 3 # First call's argument

print("Argument for call to f is ", y)

print("f(y) result is ", f(x=y))

print("Argument y on return from f is ",y," with id ",id(y),"\n")

y = [1,2,3] # Second call's argument that assigns a new list to the parameter

print("Argument for call to f is ", y)

print("f(y) result is ", f(x=y))

print("Argument y on return from f is ",y," with id ",id(y),"\n")

z = [5,7] # g argument is another list

print("Argument for call to g is ", z," with id ",id(z))

print("g(z) result is ", g(x=z))

print("Argument z on return from g is ",z," with id ",id(z),"\n")

print("End execution of passArgumentsByAssignment.py")

====================== RESTART: C:\Users\mitchell\148\_f22\passArgumentsByAssignment.py ======================

Argument for call to f is 3

Start f execution; x is 3 and its id is 2521261998384

Finishing f execution; x is -107 and its id is 2521301724688

f(y) result is None

Argument y on return from f is 3 with id 2521261998384

Argument for call to f is [1, 2, 3]

Start f execution; x is [1, 2, 3] and its id is 2521272223744

Finishing f execution; x is [1, 2, 3, [0, 0, 0]] and its id is 2521272223744

f(y) result is None

Argument y on return from f is [1, 2, 3, [0, 0, 0]] with id 2521272223744

Argument for call to g is [5, 7] with id 2521262818624

Start g execution; x is [5, 7] and its id is 2521262818624

Finishing g execution; x is [7, 5] and its id is 2521262818624

g(z) result is None

Argument z on return from g is [7, 5] with id 2521262818624

End execution of passArgumentsByAssignment.py

Namedtuple

*In large-scale software development*, a Python function’s returned tuple, such as (-1,6), in itself, is not very informative.

Someone testing such code and looking at this returned value would not know if (-1,6) was a “good or bad” return value …

The cryptic display of a returned tuple can cause a) confusion/errors when extracting tuple values back in the calling code and

b) readability/understandability issues, especially in large software systems

The a) and b)concerns are the motivation for Python’s namedtuple construct.

*A namedtuple is an immutable Python type that combines useability of a dict and the clarity of OO “dot” notation.*

Python's namedtuple was created to *improve code readability* by providing a way to access values using descriptive field names and the “dot” syntax, instead of integer indices that do not provide contextual info on the value meanings. This also makes the code easier to maintain.

Simple example

>>>from collections import namedtuple

… meal = namedtuple("meal",["food","beverage","dessert"])

… type(meal)

… <class 'type'> < -- A nametuple is a relatively simple Python type

… breakfast = meal("breakfast burrito","black coffee","cinnamon role") *< - - An instance of type meal; it’s a “Heart attack in a bag”*

… print(breakfast.beverage) # Common “dot” accessor notation now makes it clear what the printed value means

… black coffee

Demo: ntuple.py in the homepage Python3 subdirectory

**Name scope**

The **scope of a name** is the region of a program in which that name has meaning.

If your code refers to the name x, then Python searches for x in the following namespaces in order:

1. Local: If you refer to x inside a function, then the interpreter first searches for it in the innermost scope that’s local to that function.
2. Enclosing: A def statement’s code can itself contain def statements. If x is not in the local scope but appears in a function that is defined inside another function, then the interpreter searches in the enclosing function’s scope.
3. Global: If neither of the above searches succeeds, then the interpreter looks in the global scope next.
4. Built-in: If searches 1., 2., and 3. fail, then the interpreter tries the built-in scope.

This is the “LEGB (Local/Enclosing/Global/Built-in) rule” as commonly called in Python literature (although the term does not appear in the [Python documentation](https://docs.python.org/3)).

Namespace lookup examples:

>>> x = 'global'

2

3>>> def f():

4...

5... def g():

6... print(x)

7...

8... g()

9...

10

11>>> f()

12global

Line 6 can refer to only one possible x. It displays the object value defined in the global namespace, referred to as 'global'.

>>> x = 'global'

2

3>>> def f():

4... x = 'enclosing'

5...

6... def g():

7... print(x)

8...

9... g()

10...

11

12>>> f()

13enclosing

As in the previous example, g() refers to x. But this time, it has two definitions to choose from:

**Line 1** defines x in the global scope.

**Line 4** defines x again in the enclosing scope.

According to the LEGB rule, the interpreter finds the value from the enclosing scope before looking in the global scope. So the print() statement on **line 7** displays 'enclosing' instead of 'global'.

>>> x = 'global'

2

3>>> def f():

4... x = 'enclosing'

5...

6... def g():

7... x = 'local'

8... print(x)

9...

10... g()

11...

12

13>>> f()

14local

Now the print() statement on **line 8** has to distinguish between three different possibilities:

**Line 1** defines x in the global scope.

**Line 4** defines x again in the enclosing scope.

**Line 7** defines x a third time in the scope that’s local to g().

Here, the LEGB rule dictates that g() sees its own locally defined value of x first. So the print() statement displays 'local'.

When all searches for an x definition fail, a statement such as print(x) throws an error.

**Namespace Dictionaries**

Python implements the globals and locals namespaces as dictionaries.

Note: The built-in namespace does not behave like a user-defined dictionary, and is implemented as a module.

Python provides built-in functions called globals() and locals() for accessing global and local namespace dictionaries.

The built-in function globals() returns a reference to the current global namespace dictionary. You can use it to access the objects in the global namespace. An example of what it looks like when the main program (having no docString) starts:

>>> type(globals())

<class 'dict'>

>>> globals()

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, etc.} *< = Complete contents depend on Python version*

The results of globals() and/or locals() calls changes when a name is defined in either scope. Example:

>>> x = 'foo'

>>> globals()

{'\_\_name\_\_': '\_\_main\_\_', '\_\_doc\_\_': None, '\_\_package\_\_': None, . . . , 'x': 'foo'}

**Classes**

All Python objects are represented by *classes*. That is, each object is an instance of an existing class, and each class has a type,

illustrated by the very minimal possible class:

class Mini(object):

pass *< -- Class body consisting only of the pass statement is a dummy class (presumably to be expanded in*

*subsequent editing of the file in which Mini occurs*

x = Mini() *< -- In Python, a built-in class function \_\_new\_\_ creates each class instance. Unlike Java, for example, \_\_new\_\_()*

*executes automatically and does not need to be called by user code.*

*After the assignment, x is an instance of class Mini*

*Mini() behaves like a function call, and in this simple example, instance x initially has no instance attributes.*

type(x)

<class '\_\_main\_\_.Mini'>

In Python3, each class statement that introduces a class definition should inherit from object.

(doing this ensures that various built-in tools are automatically available):

class Myclass(object):

… *< -- class body*

Assuming Myclass exists

A class name followed by class arguments, for example: mc = Myclass(p1=a1, … , pn=an)

creates an instance named “mc” of the class named Myclass.

The arguments aj are passed to a user-defined class function named \_\_init\_\_ ; \_\_init\_\_ signature is:

\_\_init\_\_(self,\*, p1, … , pn)

Notice that the first signature item is NOT the \* character as in function signatures for keyword-only parameters.

self is an implicit reference to each class instance, it must always be the FIRST signature item for any class method,

not just for \_\_init\_\_.

The rest of the signature of each class function follows the same syntax as for functions defined outside of the scope of a class: \*,firstParameter, …

After the statement mc = Myclass(p1=a1, … , pn=an) is executed, the instance mc already exists.

AND

a reference to the created instance (traditionally named “self”) ALREADY exists.

*Note1 - “self” could, in theory, be any identifier, but Python programmers traditionally expect the name “self”, just as*

*simpy programmers expect the name “env” as a reference to a simpy environment*

*Note2 – when a class function f is called, the calling code DOES NOT need to provide an argument*

*for f’s signature item “self”*

All the information about an object, including its values and methods, are referred to as the *attributes* of a class.

(The term “property” has a very specific technical meaning/use in Python that will not be covered.)

\_\_init\_\_() is a user-written function for the purpose of passing initial instance arguments to a newly-created class instance.

\_\_init\_\_ is optional, but usually coded by the user for customizing class instances

Contrary to common mis-understanding, \_\_init\_\_() is NOT a class constructor; instead, \_\_init\_\_ is called as part of instance creation.

\_\_init\_\_() arguments can be arbitrary objects. There are double underscores surrounding ‘init’ (the underscores explained later)

Following the CSC148 standard of keyword-only parameters p1, …, pn, the signature for \_\_init\_\_ is: (self, \*, p1, … , pn)

An instance named dc1 of a class named DemoClass is created by the assignment dc1 = DemoClass(p1=a1, … , pn=an)

where the aj are arguments passed to the DemoClass \_\_init\_\_() function that initialize attributes of dc1.

The script demoClass.py illustrates all of the above.

[ Backgrounder: The concept of a Python class is analogous to the implementation of a struct in low-level languages such as C.

In CPython (the Python implementation dialect written in C that is used in CSC148) C structs directly implement classes.

All objects in Python are pointers to a C language struct named PyObject ]

**Subclasses**

Class B inherits from class A using the class statement syntax: class B(A):

(The above assumes that A inherited from object)

In OO languages, a subclass inherits everything from its parent class

This section is not an in-depth discussion of sub-classing.

Instead, we focus on how subclasses introduce a complication concerning testing for the type/class of an object.

This is important because the test in IF, WHILE, and other statements can be affected by use of type VS isinstance.

Below is an example

"""typeVSisinstance\_demo.py"""

class mylist(list):

"""Even without adding methods or data structures, mylist is a subclass/ of list"""

pass

mylist1 = mylist(instanceName='mylist1') # An instance of class mylist

# Because subtypes are not checked, isinstance returns incorrect results

print("Using type(mylist1) is list: Is instance mylist1 type list? ",type(mylist1) is list)

print("Using type(mylist1) is mylist: Is instance mylist1 type mylist? ",type(mylist1) is mylist)

print("Using isinstance(mylist1,list): Is mylist1 an instance of list? ", isinstance(mylist1,list))

print("The last result shows that isinstance does NOT do a subtype check, thus incorrectly comparing types")

Execution results

====================== RESTART: C:/Users/mitchell/148\_f23/Python3.6/typeVSisinstance\_demo.py =====================

Using type(mylist1) is list: Is instance mylist1 type list? False

Using type(mylist1) is mylist: Is instance mylist1 type mylist? True

Using isinstance(mylist1,list): Is mylist1 an instance of list? True

The last result shows that isinstance does NOT do a subtype check, thus incorrectly comparing types