Because learning changes everything.®

TITLE: Python Programming: Problem Solving, Packages and Libraries

Edition

Lecture PPT: Chapter 5: **Function (More advanced concepts)**

Learning Objectives

- Pass variables to a function in two different ways, that is, “pass by value” versus “pass by reference”.
- Understand concepts of Function arguments (default arguments, positional arguments and keyword arguments.)
- Explain how the Python interpreter searches for modules.
- Differentiate between recursion and iteration.
- Use “variable number of arguments” in a function definition/ call.
- List the uses of zip, lambda, map and filter in functions.
- Outline the concept of a “generator” function and the use of the “yield” statement.
- Implement the next() function or __next__() method in generators.

Call by value *vs* Call by reference

In most programming languages, there are two common methods of passing of variables between function call and function definition as follows:

- Call by value: Here, a copy of the variable value is given to the function call. So any modification to this value will not affect the original value.
- Call by reference: Here, both the original variable and the variable name inside the function call point to the same object. Hence, any modification to the object pointed to by the variable, will reflect even outside the function call. So one could say "a reference" to the original object is passed or that the "address" of the object is passed. So here, any modification done to the object within the function call will automatically modify the original object.

In Python, both "call by value" and "call by reference" are possible. Which particular scheme will be used depends upon the "type of object" being passed to a function call as parameter.

Function parameters versus arguments

Parameters

- parameters are the name within the function definition or function declaration.
- A parameter is a temporary variable that a function uses to receive a value in order to run the code.
- So a parameter is found in a **function definition**.
- So a function parameter will remain the same

Arguments

- An argument on the other hand, is the value passed to the function. It supplies the value to the parameter of the function.
- So an argument is used in a **function call.**
- However a function argument may change with each call

Important terms relating to function arguments

The three important terms relating to function arguments are:

- Default arguments
- Positional arguments
- Keyword arguments

Default arguments.
(How the scheme works)

- In the function definition an argument may be provided a default value.
- Then when that particular function is being called, the caller has the option of giving or not giving a value for that parameter.
- In case the function caller does not give any value for this particular parameter (Whose default value has been provided in the function definition), then the default value is used in that particular function call.
- However in case the function caller, provides a value for that particular parameter, then the default value present in the function definition is ignored and instead the value provided by the function caller is used.
- So “default arguments” are always provided in a function definition and not in a function call.

Rules for default values

- The default value assigned to the parameter should be a constant only. This means that you cannot assign a variable as a default value.
- Only those parameters, which are at the end of the list can be given default values.
- When you provide "default values", you must start from the "right most" parameter.
- So you cannot have a scenario where you provide a default value to some parameter but don't provide a default value to a parameter which is to the "right" of this parameter.
- So in the list of parameters in the function definition, starting from the left, you must first have only those parameters "without default values" and then have only those parameters "with default values".
- You cannot have a parameter "without default value" to the right of a "parameter with default value".
- This means that you cannot assign a default value to a variable if you have not assigned a default value to a variable to its right.

Advantages of providing “default values” to parameters in a function definition

- If a default value is provided in a function definition, then you have the option of providing a value or not providing a value in a function call.
- If you provide a value to the argument in the function call then this value will be taken, but if you do not provide a value, then the default value is taken.

Differences between a function definition and a function call

	Function definition	Function call
1	It is that part of the script which defines the function. So you are required to use the "def" keyword when defining a function.	It is that part of the function where you "use or call" the function. Here, you simply "call or use" the function.
2	The "names or labels" used for the parameters in the function definition are called "keywords"	In a function call, you may "use" the keywords of the function definition to specify as to which parameter in the function call refers to which parameter in the function definition. This could be necessary in a scenario where you are not "passing" the parameters from "function call" to "function definition" in the "same order" as in the function definition.
3	If you want to provide "default" values to some parameters, then you may do so in the function definition	In a function call you have the option of not providing any value to those parameters for which default values have been provided in the function definition.

What are “keywords”?

- In a function definition, each argument is given a “label or a name”.
- These labels in the function definition, not the function call are called “keywords”.
- So “keywords” relate to function definition.

Passing argument from function call to function definition

The two important methods regarding how parameters are passed between the caller and the called are:

- Match the arguments by position.
- Match arguments by name, or matching by “key-value” pairs.

Note: When you call a function, the function definition will normally expect two things:

- The function definition normally expects as many arguments in the function call as there are in the function definition. The exception to this is of course the scenario when the function definition provides default values (starting from right) for some or all of the arguments.
- The function definition assumes that the ‘order’ in which the arguments are provided corresponds to the order in which they are in the function definition.
- The “order” may not be preserved if you are providing the arguments in form of “key-value” pairs.

Matching by position

Matching by position is best understood by an example.
Consider the example given below (From the book):

```
def f(a,b,c='cat', d= 'dog'):
    pass
f('ant', 'bee') # OK
f('A', 'B', 'C') #OK
f(1,2,3,4) #OK
f(1) # ERROR because you must provide at least 2 arguments ie for a and b
f() # Error
```

- **The function f() in its definition takes 4 parameters a, b, c and d.**
- **In a function call the "order" of the parameters must be preserved. You cannot change the "order" of the arguments. For example you cannot provide the argument for parameter "b" before that for "a".**
- **The function call must provide a minimum of 2 and a max of 4 arguments.**
- **If you provide less than 2 or more than 4 arguments, there will be an error.**
- **The above script is an example of a function call by position.**
- **Detailed explanation of the code is given in the book**

The need for having “keyword” arguments. (Also called “key-value” pair arguments

Key-value pair passing of arguments is available in Python to cater to a scenario, when you

- **don’t follow the “order” of passing the arguments,**
- **or you want to skip some parameters in between and then pass those at the end of the list.**

In some programming languages, this is possible.
For instance, in some programming languages (Like Visual Basic) you can use the syntax f(a, , b,c) where the two consecutive commas indicate a missing parameter.
You cannot do this in Python.

How "keyword arguments" or "key-value pair" are used in Python

- In Python, a technique called "keyword arguments" is used instead.
- In this scheme, while "passing" the arguments from the function call to the function definition, one uses the "keywords" or the "names" of the arguments in the function definition to tell the function definition which parameter from the function call is linked to which parameter in the function definition.
- However in order to use the key-value pair, you must know the "names" of the parameters used in the function definition.
- So when you are using third party libraries/ modules, you may have to at times see the source code file to see the names of parameters in function definition.
- Off course a well written library will provide you these names of parameters through __doc__ attribute, so that you don't kneed to see the source code. (The __doc__ attribute is covered later).

So in Conclusion:

- The default values are always provided in the function definition, not in the function call.
- The ‘keyword-argument’ pairs are always provided in the function call and not in the function definition.
- The ‘keywords’ used in the keyword-argument’ pairs must be same as defined in the function definition.

Using both “default-values” and “keyword-arguments” together

It is possible to have both “default-values” and “keyword-arguments” together.
The following example (From book) clarifies the concept

```
1   def f1(a, b = 'BOY', c= 'CAT'): # Default values to b and c
2       print('a->', a, 'b->', b, 'c->', c)
3
4   f1('apple') #OK. Will use default values for b and c
5   f1(a= 'ant') # Also OK
6   f1(b = 'baby', a ='ass') # Will use default for 3rd parameter ie c = ‘ÇAT’
7   # . . . OUTPUT IS . . .
8   a-> apple b-> BOY c-> CAT
9   a-> ant b-> BOY c-> CAT
10  a-> ass b-> baby c-> CAT
```

Explanation is given in the book

- **In Line 4, default values for parameters “a” and “b” are used.**
- **In Line 5 and 6, the “keyword- argument” scheme is used.**

Variable number of arguments in a function call by with * in function definition

- In the function definition, you use the format f(*arg) where ‘f’ is the name of the function you are defining, and *arg is a variable list of arguments that this function call can expect.
- In the function call, you use it like any other function call except that you may vary the number of arguments to the function when calling the function.
- So you could use syntax f(1,2,3) or f(1,2,3,4,5), and so on for function call and the Python interpreter will understand that these are two different function calls on the same function, but with different number of parameters.
- So when you call the function using say f(1,2,3), you are providing three parameters, that is,1,2 and 3. Also, when you use f(1,2,3,4,5), you are providing five parameters to the function when calling the function.

Example of:- Variable number of arguments in a function call by with * in function definition

The book example reproduced below, shows a function f(*arg) which can accept “variable number” of arguments.

```python
def f(*arg): # def has *arg. Can give variable numbers of arguments.
    total = 0
    for x in arg:
        total = total + x
    return total
print (f(1,2,3,4)) # Call the function with 4 arguments
print(f(1,2))       # Call the function with 2 arguments
```

```
# Output
10
3
```

Using **kwarg in function definition to pass a key worded, variable-length of arguments.

In the function definition:

- the format **kwarg is used in the function definition and not in the function call.

In the function call:

- You are passing a variable length of arguments during function call.
- You are passing the arguments as a “pair” of “variable name” and its “value”. So **kwarg differs from *arg that in *arg you provide only one value for each argument, whereas in **kwarg you provide two values, one for the key and other for its value.
- Also, the key-value pair is being passed to the function as a “dictionary”. So a dictionary with variable name ‘kwarg’ will be created just like any other dictionary in Python and one can use any of the attributes or methods of this dictionary. Further, also note that since ‘kwarg’ is a dictionary, there is no “order” of the items in it. Also, the variable name ‘kwarg’ is just by convention, one can use any valid Python name.

Within a function definition *arg acts like a tuple and **kwarg acts like a dictionary.

The following example (From book) shows that within a function definition

- ***arg acts like a tuple and**
- ****kwarg acts like a dictionary**

```
1    def f1(*arg):
2        print("arg is", arg)
3        print("Type of arg is ", type(arg))
4
5    def f2(**kwarg):
6        print("kwarg is", kwarg)
7        print("Type of kwarg is ", type(kwarg))
8
9    f1(1,2,3)
10   f2(a = 1, b = 2, c = 3)
```

```
11   # Output
12   arg is (1, 2, 3)
13   Type of arg is  <class 'tuple'>
14   kwarg is {'b': 2, 'a': 1, 'c': 3}
15   Type of kwarg is  <class 'dict'>
```

The above script is self-evident. The output from line 2 and 3 are lines 12 and 13, which show that *arg is a tuple. Similarly, the output from lines 6–7 are lines 14-15, which show that **kwarg is a dictionary.

Additional note:
How the Python interpreter searches for modules

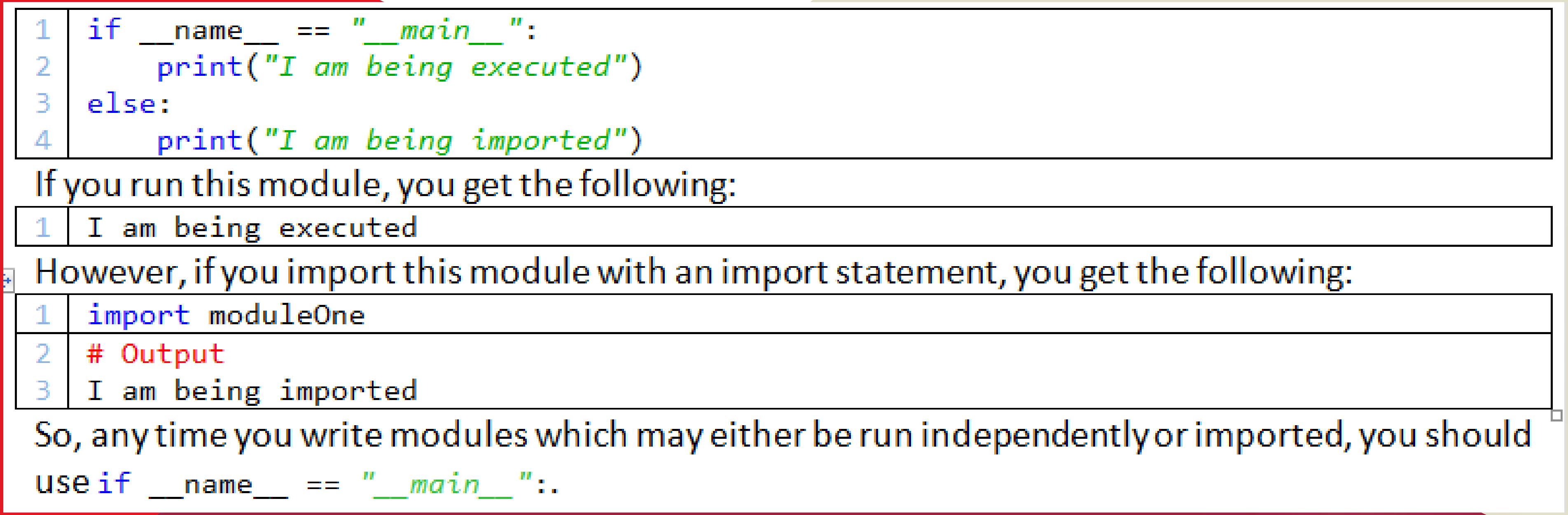
The Python interpreter searches the module in the following sequence:

- **It searches for the module in the current directory.**
- **It then searches for a module are given in the sys.path attribute of the sys module.**
- **If the module is not found in the paths contained in the list maintained by sys.path(), then the Python interpreter looks for it in the PYTHONPATH environmental variable.**
- **If the Python interpreter does not find the module by any of the above methods, then it will throw an error.**

Using if __name__ == "__main__":

(Testing whether the script is being run directly or being imported by something else.)

In Python, every file (ending with .py) can be executed directly or can be imported by another module.
The following screen shot is from an example in the book. It shows how __name__ = “__main__” may be used:



```python
if __name__ == "__main__":
    print("I am being executed")
else:
    print("I am being imported")
```

If you run this module, you get the following:

```
I am being executed
```

However, if you import this module with an import statement, you get the following:

```python
import moduleOne
# Output
I am being imported
```

So, any time you write modules which may either be run independently or imported, you should use `if __name__ == "__main__":`.

Recursion

- **Suppose a function "calls itself" repeatedly and in each call, "breaks" the problem into "simpler cases". Such a function is a "recursive function".**
- **There is another additional requirement —the function must "terminate" at some point of time or else it would be an infinite loop.**
- **A recursive function will terminate only if:**
  - In each iteration, that is, each call, the problem being solved becomes "smaller".
  - There is a "base case". A "base case" occurs where the problem gets solved without further recursion or without any further call to the function. It is the "base case" which ends the recursion. Without a base case, the script would go into an unending "infinite loop".

Characteristics of a Recursive algorithm

All recursive algorithms must have three characteristics:

- **A recursive algorithm must call itself, recursively.**
- **A recursive algorithm must have a base case. A base case occurs when the problem after repeated calls to the recursive function, becomes so small that it can be solved directly.**
- **A recursive algorithm after each call to the recursive function, must move toward the base case. So, in a well-designed recursive algorithm, after each iteration, the problem must become "smaller".**

The following pseudo-code (From book) shows how a recursive algorithm works

```
def recursiveFunction(attributes):
    if (test for some_simple_case):
        return (Simple computation without recursion)
    else:
        return recursive_solution
```

Using recursion for finding factorial of a number

- **The factorial of a positive integer 'n' is mathematically represented as n! and can be defined as: n! = n*(n-1)*(n-2) ….. 3*2*1.**
- **Further, the factorial of 0 is defined as: 0! = 1**

So, you can write n! or fact(n) as follows:

$$fact(n) = \begin{pmatrix} 1 & if\ n = 0 \\ n * fact(n-1) & if\ n > 0 \end{pmatrix}$$

- **Notice that in the above equation, fact(n) = n* fact(n-1).**
- **So, the initial problem of finding fact(n) has been broken down into finding fact(n-1).**
- **Therefore, recursion is a good technique for finding fact(n).**

Additional soved examples of use of recursion

- **Recursive function to find a number is even or not**
- **Recursive function to find a^b**
- **Recursive function to find GCD (Greatest Common Divisor) of two numbers using the Euclidean algorithm.**
- **Recursive function to generate Fibonacci numbers**

Recursion versus iteration (Advantage/ disadvantage of recursion)

Advantages of recursion

- Many recursive algorithms are very elegant, and hence, produce clean code.
- Recursion can often break down a complex task into simpler sub-problems.
- Many sequences are easier to generate via recursion than via iteration.

Disadvantages of recursion

- At times, a recursive algorithm may be difficult to understand.
- Some recursive algorithms take up a of resources (computing time and memory).
- Many recursive functions may be hard to debug.

Tower of Hanoi

The book provides a recursive solution to the tower of Hanoi problem. Can you do it iteratively?

Hint:- As per this web site:- http://hermes.di.uoa.gr/exe_activities/algorithmoi/_3.html

The algorithm is:-

Simpler statement of iterative solution

Alternating between the smallest and the next-smallest disks, follow the steps for the appropriate case:

For an even number of disks:

- make the legal move between pegs A and B
- make the legal move between pegs A and C
- make the legal move between pegs B and C
- repeat until complete

For an odd number of disks:

- make the legal move between pegs A and C
- make the legal move between pegs A and B
- make the legal move between pegs B and C
- repeat until complete

In each case, a total of 2^n-1 moves are made.

Memoizing

- It is an “optimization” technique.
- Results of function calls are stored or “cached”.
- So, before doing calculation in a function call, the stored values are checked to see if the result is available in the store or “cache”.
- If the result is present in the “cache”, then this value is used.
- If the result is not present, then it is calculated and at the same time stored for future use.

Memoizing may be particularly useful in recursion, where intermediate results may be needed “again and again”.

zip function is an "aggregator" meaning that it aggregates 2 or more iterables or sequences. The individual items of the iterables are combined to form tuples. The zip() function stops whenever the shortest of the iterables is exhausted.

The syntax of the function is:-

```
zip(*iterables)
```

The iterables can be containers like lists, tuples, strings etc

Using zip to “unzip”

A common confusion with beginners is that the zip function can also be used for “unzipping”.

So note the following:-

- When the zip function is provided with iterables (Think of iterables as some sequence), then it will zip them
- But if the zip function is provided with a star ie (*) followed by a zipped object, then it will “unzip” it

Example code shows how the zip() function is used to zip and unzip

```
1   caps = ['A', 'B', 'C', 'D']
2   smalls = ['a', 'b', 'c', 'd']
3   # Zip the above 2 lists
4   zip_result = zip(caps, smalls)
5   # Cast the zipped object to a list
6   zip_list = list(zip_result)
7   # Check that we have a zipped list of 2 lists
8   print('zip_list->', zip_list)
9   # Unzip the zipped list
10  cap_list, small_list = zip(*zip_list)
11  print('cap_list->', cap_list)
12  print('small_list->', small_list)
```

```
13  # OUTPUT
14  zip_list-> [('A', 'a'), ('B', 'b'), ('C', 'c'), ('D', 'd')]
15  cap_list-> ('A', 'B', 'C', 'D')
16  small_list-> ('a', 'b', 'c', 'd')
```

Line 4: zip_result = zip(caps, smalls). This causes the 2 lists to get zipped.
Line 10: cap_list, small_list = zip(*zip_list). This causes the 2 lists zipped earlier in line 4 to get unzipped

Lambda functions

In Python, you can write an “anonymous” function, that is, a function without a name. Note that a typical Python function (Or method) begins with the keyword ‘def’. However, a lambda function does not have a name. The syntax of lambda function is as follows:

```
lambda arguments: expression
```

The following code (From book) shows use of a normal function and then a lambda function to calulate the square of a number.

```python
# A normal function which squares a number
def square_numb(x):
    return x ** 2

print(square_numb(10))

# A lambda function for squaring a Number
get_square = lambda x: x ** 2
print(get_square(20))
```

When to use the lambda function

- Lambda functions are generally used where you need a nameless function for a short period or for a small calculation.
- Lambda functions are also used as an argument to a higher-order function (A higher order function is one which takes as its argument the result of another function.) In python, lambda functions are often used in combination with other functions, such as map() and filter().

map() function

map() function takes a function and a sequence as its arguments. It returns an iterator (Iterators are discussed later, but for the present, you can think of an iterator as some kind of a sequence over which you can “iterate” one by one). The syntax of map() is:-

```
# First argument is function, second is an iterable
map_obj = map(function, iterable)
# There can be more than 1 iterable also
map_obj = map(function, iterable1, iterable2, iterable3..., iterableN)
```

The following example code shows how the map() function may be used

```
# Define a function which gives cube of a number
def cube_numb(n):
    return n ** 3
# Create a list of numbers
numb_list = [1, 2, 3, 4, 5]
# Use map() to generate cubes of numbers in the list
cube_seq = map(cube_numb, numb_list)
# cube_seq is an object of map class. It is not a list
print(type(cube_seq))
# But you can cast a map object to list
print(list(cube_seq))
```

```
# Output
<class 'map'>
[1, 8, 27, 64, 125]
```

filter() function

- **filter() is a function to remove False items from a sequence.**
- **A filter() function takes another function as its first argument and a sequence (Or rather an iterable) as its second argument.**
- **The first argument, that is, the function must return a Boolean value, that is, True or False.**
- **Syntax of filter() is:-**

```
# function must return a boolean True or False
filter(function, sequence)
```

Following code shows how filter may be used to get filter out odd numbers from a list. (Note that filter() function does not return a list object. If you need a list object, you need to cast it into a list:-

```
my_list = [ x for x in range(10)]
list_odds = filter(lambda x: x%2 == 0, my_list)
print(list(list_odds))
# Output
[0, 2, 4, 6, 8]
```

Generator functions

To understand a generator function, first consider what an “ordinary” function does:-

- When you call an ordinary Python function, it starts execution till it meets a return statement. (Of course, if there is no return statement then implicitly a None is returned).
- After the function execution is over, the control returns back to the main program and any values or temporary work done by the function is lost.
- So, a new call to the function causes everything to start from scratch.

However, in Python it is also possible to write a “generator function” which stores or saves “intermediate” values and is able to provide them through “iterations or steppings”.

- Here one can use a “yield” statement. On seeing a “yield” statement, the interpreter understands that the function is not a “normal” function but rather it is a “generator” function.
- In a “normal” function, the function returns a value when it sees the keyword “return”.
- However, a generator function returns a value when it sees the keyword “yield”. Though a generator function is also allowed to have ‘return’ statement, but in order to become a generator function, it must have a ‘yield’ statement.
- If the body of a function has a “yield” statement, the function, becomes a generator function even if it also has a “return” statement.

Advantages of using generators

- **Generator functions use less memory and consume less time. So their performance is generally better than a “normal” function.**
- **With generator functions you can “save the state” between two “calls” to a function.**

The text book gives an excellent example of use of generator functions to calculate prime numbers starting from number 10

Thank You!

For any queries or feedback contact us at:

support.india@mheducation.com

1800-103-5875

www.mheducation.co.in