

Python - understanding Python function basics

In this notebook, we will learn basic elements of defining a function. All of content below are directly reference through standard Python documentation referring to [sections 4.7 & 4.8](#).

4.7. Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
In [ ]: def fib(n):    # write Fibonacci series up to n
        """Print a Fibonacci series up to n."""
        a, b = 0, 1
        while a < n:
            print(a, end=' ')
            a, b = b, a+b
        print()
```

```
In [ ]: # Now call the function we just defined:
        fib(2000)
```

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597

The keyword `def` introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or docstring. (More about docstrings can be found in the section Documentation Strings.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The execution of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables and variables of enclosing functions cannot be directly assigned a value within a function (unless, for global variables, named in a global statement, or, for variables of enclosing functions, named in a nonlocal statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using call by value (where the value is always an object reference, not the value of the object). 1 When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

A function definition associates the function name with the function object in the current symbol table. The interpreter recognizes the object pointed to by that name as a user-defined function. Other names can also point to that same function object and can also be used to access the function:

```
In [ ]: print(fib)
        f = fib
        f(100)
```

```
<function fib at 0x0000021604F12200>
0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that fib is not a function but a procedure since it doesn't return a value. In fact, even functions without a return statement do return a value, albeit a rather boring one. This value is called None (it's a built-in name). Writing the value None is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using print():

```
In [ ]: fib(0)
        print(fib(0))
```

None

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
In [ ]: def fib2(n): # return Fibonacci series up to n
        """Return a list containing the Fibonacci series up to n."""
        result = []
        a, b = 0, 1
        while a < n:
            result.append(a)    # see below
            a, b = b, a+b
        return result

        f100 = fib2(100)    # call it
        f100                # write the result
```

```
Out[ ]: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

The return statement returns with a value from a function. return without an expression argument returns None. Falling off the end of a function also returns None.

The statement result.append(a) calls a method of the list object result. A method is a function that 'belongs' to an object and is named obj.methodname, where obj is some object (this may be an expression), and methodname is the name of a method that is defined by the object's type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own

object types and methods, using classes, see Classes) The method `append()` shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to `result = result + [a]`, but more efficient.

4.8. More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined. 4.8.1. Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
In [ ]: def ask_ok(prompt, retries=4, reminder='Please try again!'):
        while True:
            ok = input(prompt)
            if ok in ('y', 'ye', 'yes'):
                return True
            if ok in ('n', 'no', 'nop', 'nope'):
                return False
            retries = retries - 1
            if retries < 0:
                raise ValueError('invalid user response')
            print(reminder)
```

This function can be called in several ways:

- giving only the mandatory argument: `ask_ok('Do you really want to quit?')`
- giving one of the optional arguments: `ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the `in` keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the defining scope, so that

```
In [ ]: i = 5

        def f(arg=i):
            print(arg)

        i = 6
        f()
```

5

will print 5.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
In [ ]: def f(a, L=[]):
        L.append(a)
        return L

print(f(1))
print(f(2))
print(f(3))

# This will print
# [1]
# [1, 2]
# [1, 2, 3]
```

```
[1]
[1, 2]
[1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
In [ ]: def f(a, L=None):
        if L is None:
            L = []
        L.append(a)
        return L
```

4.8.2. Keyword Arguments

Functions can also be called using keyword arguments of the form `kwarg=value`. For instance, the following function:

```
In [ ]: def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
        print("-- This parrot wouldn't", action, end=' ')
        print("if you put", voltage, "volts through it.")
        print("-- Lovely plumage, the", type)
        print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
In [ ]: parrot(1000)                                # 1 positional argument
        parrot(voltage=1000)                          # 1 keyword argument
        parrot(voltage=1000000, action='VOOOOOM')      # 2 keyword arguments
        parrot(action='VOOOOOM', voltage=1000000)      # 2 keyword arguments
        parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
        parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
# parrot()                # required argument missing
# parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
# parrot(110, voltage=220)    # duplicate value for the same argument
# parrot(actor='John Cleese') # unknown keyword argument
```

```
-- This parrot wouldn't vroom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't vroom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
-- This parrot wouldn't jump if you put a million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bereft of life !
-- This parrot wouldn't vroom if you put a thousand volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's pushing up the daisies !
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. actor is not a valid argument for the parrot function), and their order is not important. This also includes non-optional arguments (e.g. parrot(voltage=1000) is valid too). No argument may receive a value more than once. Here's an example that fails due to this restriction:

```
In [ ]: def function(a):
```

```
    pass
```

```
function(0, a=0)
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[12], line 6
      1 def function(a):
      3     pass
----> 6 function(0, a=0)

TypeError: function() got multiple values for argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see Mapping Types — dict) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `name` (described in the next subsection) which receives a tuple containing the positional arguments

beyond the formal parameter list. (name must occur before ****name**.) For example, if we define a function like this:

```
In [ ]: def cheeseshop(kind, *arguments, **keywords):
        print("-- Do you have any", kind, "?")
        print("-- I'm sorry, we're all out of", kind)
        for arg in arguments:
            print(arg)
        print("-" * 40)
        for kw in keywords:
            print(kw, ":", keywords[kw])
```

It could be called like this:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Note that the order in which the keyword arguments are printed is guaranteed to match the order in which they were provided in the function call.

4.8.3. Special parameters

By default, arguments may be passed to a Python function either by position or explicitly by keyword. For readability and performance, it makes sense to restrict the way arguments can be passed so that a developer need only look at the function definition to determine if items are passed by position, by position or keyword, or by keyword.

A function definition may look like:

```

math
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    -----
    |           |           |
    |           | Positional or keyword |
    |           |           |
    |           |           | - Keyword only
    -- Positional only

```

where / and * are optional. If used, these symbols indicate the kind of parameter by how the arguments may be passed to the function: positional-only, positional-or-keyword, and keyword-only. Keyword parameters are also referred to as named parameters. 4.8.3.1.

Positional-or-Keyword Arguments

If / and * are not present in the function definition, arguments may be passed to a function by position or by keyword. 4.8.3.2. Positional-Only Parameters

Looking at this in a bit more detail, it is possible to mark certain parameters as positional-only. If positional-only, the parameters' order matters, and the parameters cannot be passed by keyword. Positional-only parameters are placed before a / (forward-slash). The / is used to logically separate the positional-only parameters from the rest of the parameters. If there is no / in the function definition, there are no positional-only parameters.

Parameters following the / may be positional-or-keyword or keyword-only.

4.8.3.3. Keyword-Only Arguments

To mark parameters as keyword-only, indicating the parameters must be passed by keyword argument, place an * in the arguments list just before the first keyword-only parameter.

4.8.3.4. Function Examples

Consider the following example function definitions paying close attention to the markers / and *:

```
In [ ]: def standard_arg(arg):
        print(arg)

def pos_only_arg(arg, /):
    print(arg)

def kwd_only_arg(*, arg):
    print(arg)

def combined_example(pos_only, /, standard, *, kwd_only):
    print(pos_only, standard, kwd_only)

# The first function definition, standard_arg, the most familiar form, places
# no restrictions on the calling convention and arguments may be passed by position

print(standard_arg(2))
print(standard_arg(arg=2))

# The second function pos_only_arg is restricted to only use positional parameters

print(pos_only_arg(1))

# print(pos_only_arg(arg=1)) # error
```

```

# The third function kwd_only_args only allows keyword arguments as indicated by a
# print(kwd_only_arg(3)) # error
print(kwd_only_arg(arg=3))

# And the last uses all three calling conventions in the same function definition:

# print(combined_example(1, 2, 3)) # error

print(combined_example(1, 2, kwd_only=3))

print(combined_example(1, standard=2, kwd_only=3))

# print(combined_example(pos_only=1, standard=2, kwd_only=3)) # error

```

```

2
None
2
None
1
None
3
None
1 2 3
None
1 2 3
None

```

Finally, consider this function definition which has a potential collision between the positional argument name and `**kwds` which has name as a key:

```

math
def foo(name, **kwds):
    return 'name' in kwds

```

There is no possible call that will make it return True as the keyword 'name' will always bind to the first parameter. For example:

```

math
foo(1, **{'name': 2})

```

But using / (positional only arguments), it is possible since it allows name as a positional argument and 'name' as a key in the keyword arguments:

```

math
def foo(name, /, **kwds):

    return 'name' in kwds
```math

```



```

```math
foo(1, **{'name': 2})
True

```

In other words, the names of positional-only parameters can be used in `**kwargs` without ambiguity.

4.8.3.5. Recap

The use case will determine which parameters to use in the function definition:

```

def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):

```

As guidance:

Use positional-only if you want the name of the parameters to not be available to the user. This is useful when parameter names have no real meaning, if you want to enforce the order of the arguments when the function is called or if you need to take some positional parameters and arbitrary keywords.

Use keyword-only when names have meaning and the function definition is more understandable by being explicit with names or you want to prevent users relying on the position of the argument being passed.

For an API, use positional-only to prevent breaking API changes if the parameter's name is modified in the future.

4.8.4. Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see Tuples and Sequences). Before the variable number of arguments, zero or more normal arguments may occur.

```

math
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))

```

Normally, these variadic arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are 'keyword-only' arguments, meaning that they can only be used as keywords rather than positional arguments.

```

math
def concat(*args, sep="/"):

```

```
return sep.join(args)
```

```
concat("earth", "mars", "venus")
'earth/mars/venus'
```

```
concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```math
```

#### ## 4.8.5. Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate start and stop arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
```math
list(range(3, 6))           # normal call with separate arguments
[3, 4, 5]

args = [3, 6]

list(range(*args))          # call with arguments unpacked from a
list
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
math
def parrot(voltage, state='a stiff', action='vroom'):

    print("-- This parrot wouldn't", action, end=' ')

    print("if you put", voltage, "volts through it.", end=' ')

    print("E's", state, "!")

d = {"voltage": "four million", "state": "bleedin' demised",
     "action": "VOOM"}

parrot(**d)
```

-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !

4.8.6. Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments: `lambda a, b: a+b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```
math
def make_incrementor(n):

    return lambda x: x + n

f = make_incrementor(42)

f(0)
42

f(1)
43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

```
math
pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

pairs.sort(key=lambda pair: pair[1])

pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.8.7. Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line after the first line of the string determines the

amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace "equivalent" to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
math
def my_function():

    """Do nothing, but document it.

    No, really, it doesn't do anything.

    """

    pass

print(my_function.__doc__)
Do nothing, but document it.

No, really, it doesn't do anything.
```

4.8.8. Function Annotations

Function annotations are completely optional metadata information about the types used by user-defined functions (see PEP 3107 and PEP 484 for more information).

Annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a required argument, an optional argument, and the return value annotated:

```
math
def f(ham: str, eggs: str = 'eggs') -> str:

    print("Annotations:", f.__annotations__)

    print("Arguments:", ham, eggs)
```

```
    return ham + ' and ' + eggs
```

```
f('spam')
```

```
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>,  
'eggs': <class 'str'>}
```

```
Arguments: spam eggs
```

```
'spam and eggs'
```