Functions

All Versions

Functions in Python provide organized, reusable code to preform a set of actions. Functions simplify the coding process, prevent redundant logic, and make code easier to follow. This topic describes the declaration and utilization of functions in Python.

Examples

Defining and calling simple functions

Using the def statement is the most common way to define a function in python. This statement is a so called *single clause compound statement* with the following syntax:

```
def function_name(parameters):
    statement(s)
```

function_name is known as the *identifier* of the function. Since a function definition is an executable statement its execution *binds* the function name to the function object which can be called later on using the identifier.

parameters is an optional list of identifiers that get bound to the values supplied as arguments when the function is called. A function may have an arbitrary number of arguments which are separated by commas.

statement(s) – also known as the *function body* – are a nonempty sequence of statements executed each time the function is called. This means a function body cannot be empty, just like any *indented block* .

Here's an example of a simple function definition which purpose is to print Hello each time it's called:

```
def greet():
    print("Hello")
```

Now let's call the defined greet() function:

```
greet()
# Out: Hello
```

That's an other example of a function definition which takes one single argument and displays the passed in value each time the function is called:

```
def greet_two(greeting):
    print(greeting)
```

After that the greet_two() function must be called with an argument:

```
greet_two("Howdy")
# Out: Howdy
```

Also you can give a default value to that function argument:

```
def greet_two(greeting="Howdy"):
    print(greeting)
```

Now you can call the function without giving a value:

```
greet_two()
# Out: Howdy
```

You'll notice that unlike many other languages, you do not need to explicitly declare a return type of the function. Python functions can return values of any type via the return keyword. One function can return any number of different types!

```
def many_types(x):
    if x < 0:
        return "Hello!"
    else:
        return 0

print many_types(1)
print many_types(-1)

# Output:
0
Hello!
```

As long as this is handled correctly by the caller, this is perfectly valid Python code.

A function that reaches the end of execution without a return statement will always return None :

```
def do_nothing():
    pass

print(do_nothing())
# Out: None
```

As mentioned previously a function definition must have a function body, a nonempty sequence of statements. Therefore the pass statement is used as function body, which is a null operation – when it is executed, nothing happens. It does what it means, it skips. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed.

Defining a function with an arbitrary number of arguments

Arbitrary number of positional arguments:

Defining a function capable of taking an arbitrary number of arguments can be done by prefixing one of the argument with *

```
def func(*args):
    # args will be a tuple containing all values that are passed in
    for i in args:
        print(i)

func(1, 2, 3)  # Calling it with 3 arguments
# Out: 1
#      2
#      3

list_of_arg_values = [1, 2, 3]
func(*list_of_arg_values)  # Calling it with list of values, * expands the list
# Out: 1
#      2
#      3

func()  # Calling it without arguments
# No Output
```

You **can't** provide a default for args , for example func(*args=[1, 2, 3]) will raise a syntax error (won't even compile).

You **can't** provide these by name when calling the function, for example func(*args=[1, 2, 3]) will raise a TypeError .

But if you already have your arguments in an array (or any other Iterable), you **can** invoke your function like this: func(*my_stuff) .

These arguments (*args) can be accessed by index, for example args[0] will return the first argument

Arbitrary number of keyword arguments

You can take an arbitrary number of arguments with a name by defining an argument in the definition with **two** * in front of it:

```
def func(**kwargs):
    # kwargs will be a dictionary containing the names as keys and the values as values
    for name, value in kwargs.items():
        print(name, value)

func(value1=1, value2=2, value3=3)   # Calling it with 3 arguments
# Out: value2 2
#      value1 1
#      value3 3

func()                               # Calling it without arguments
# No Out put

my_dict = {'foo': 1, 'bar': 2}
func(**my_dict)                      # Calling it with a dictionary
# Out: foo 1
#      bar 2
```

You **can't** provide these **without** names, for example func(1, 2, 3) will raise a TypeError .

kwargs is a plain native python dictionary. For example, args['value1'] will give the value for argument value1 . Be sure to check beforehand that there is such an argument or a KeyError will be raised.

Warning

You can mix these with other optional and required arguments but the order inside the definition matters.

The **positional/keyword** arguments come first. (Required arguments).
Then comes the **arbitrary** *arg arguments. (Optional).
Then **keyword-only** arguments come next. (Required).

Finally the **arbitrary keyword** `**kwargs` come. (Optional).

```
#       |-positional-|-optional-|---keyword-only--|-optional-|
def func(arg1, arg2=10 , *args, kwarg1, kwarg2=2, **kwargs):
     pass
```

- `arg1` must be given, otherwise a `TypeError` is raised. It can be given as positional (`func(10)`) or keyword argument (`func(arg1=10)`).
- `kwarg1` must also be given, but it can only be provided as keyword-argument: `func(kwarg1=10)` .
- `arg2` and `kwarg2` are optional. If the value is to be changed the same rules as for `arg1` (either positional or keyword) and `kwarg1` (only keyword) apply.
- `*args` catches additional positional parameters. But note, that `arg1` and `arg2` must be provided as positional arguments to pass arguments to `*args` : `func(1, 1, 1, 1)` .
- `**kwargs` catches all additional keyword parameters. In this case any parameter that is not `arg1` , `arg2` , `kwarg1` or `kwarg2` . For example: `func(kwarg3=10)` .
- In Python 3, you can use `*` alone to indicate that all subsequent arguments must be specified as keywords. For instance the `math.isclose` function in Python 3.5 and higher is defined using `def math.isclose (a, b, *, rel_tol=1e-09, abs_tol=0.0)` , which means the first two arguments can be supplied positionally but the optional third and fourth parameters can only be supplied as keyword arguments.

Python 2.x doesn't support keyword-only parameters. This behavior can be emulated with `kwargs` :

```
def func(arg1, arg2=10, **kwargs):
    try:
        kwarg1 = kwargs.pop("kwarg1")
    except KeyError:
        raise TypeError("missing required keyword-only argument: 'kwarg1'")

    kwarg2 = kwargs.pop("kwarg2", 2)
    # function body ...
```

Note on Naming

The convention of naming optional positional arguments `args` and optional keyword arguments `kwargs` is just a convention you **can** use any names you like **but** it is useful to follow the convention so that others know what you are doing, *or even yourself later* so please do.

Note on Uniqueness

Any function can be defined with **none or one** `*args` and **none or one** `**kwargs` but not with more than one of each. Also `*args` **must** be the last positional argument and `**kwargs` must be the last parameter. Attempting to use more than one of either **will** result in a Syntax Error exception.

Note on Nesting Functions with Optional Arguments

It is possible to nest such functions and the usual convention is to remove the items that the code has already handled **but** if you are passing down the parameters you need to pass optional positional args with a `*` prefix and optional keyword args with a `**` prefix, otherwise args with be passed as a list or tuple and kwargs as a single dictionary. e.g.:

```
def fn(**kwargs):
    print (kwargs)
    f1(**kwargs)

def f1(**kwargs):
    print(len(kwargs))

fn(a=1, b=2)
# Out:
# {'a': 1, 'b': 2}
# 2
```

Defining a function with optional arguments

Optional arguments can be defined by assigning (using `=`) a default value to the argument-name:

```
def make(action='nothing'):
    return action
```

Calling this function is possible in 3 different ways:

```
make("fun")
# Out: fun

make(action="sleep")
# Out: sleep

# The argument is optional so the function will use the default value if the argument is
# not passed in.
make()
# Out: nothing
# Out: nothing
```

Warning

Mutable types (list , dict , set , etc.) should be treated with care when given as **default** attribute. Any mutation of the default argument will change it permanently. See Defining a function with optional mutable arguments .

Lambda (Inline/Anonymous) Functions

The lambda keyword creates an inline function that contains a single expression. The value of this expression is what the function returns when invoked.

Consider the function:

```
def greeting():
    return "Hello"
```

which, when called as:

```
print(greeting())
```

prints:

```
Hello
```

This can be written similarly as a lambda function as follows:

See note at the bottom of this section regarding the assignment of lambdas to variables. Generally, don't do it.

```
greet_me = lambda: "Hello"
```

This creates an inline function with the name greet_me that returns Hello . Note that you don't write return when creating a function with lambda. The value after : is automatically returned.

Once assigned to a variable, it can be used just like a regular function:

```
print(greet_me())
```

prints:

```
Hello
```

lambda s can take arguments, too:

```
strip_and_upper_case = lambda s: s.strip().upper()

strip_and_upper_case("  Hello   ")
```

returns the string:

```
HELLO
```

They can even take arbitrary number of arguments / keyword arguments like normal functions.

```
greeting = lambda x, *args, **kwargs: print(x, args, kwargs)
greeting('hello', 'world', world='world')
```

prints:

```
hello ('world',) {'world': 'world'}
```

lambda s are commonly used for short functions that are convenient to define at the point where they are called. For example, this line sorts a list of strings ignoring their case and ignoring whitespace at the beginning and end:

They are most often used in sorted , filter and map

```
sorted( [" foo ", "    BAR", "BaZ    ", "qux"], key=lambda s: s.strip().upper())
```

returns:

```
['    BAR', 'BaZ    ', ' foo ', 'qux']
```

==

```
sorted([" foo ", "    BAR", "BaZ    ", "qux"], key=lambda s: s.strip())
```

returns:

```
['    BAR', 'BaZ    ', ' foo ', 'qux'] # no effect of lambda, list is sorted
```

```
sorted([" foo ", "    BAR", "BaZ    ", "qux"])
```

returns:

```
['    BAR', 'BaZ    ', ' foo ', 'qux'] # list sorted.
```

```
sorted( map(lambda s: s.strip().upper(), [" foo ", "    BAR", "BaZ    ", "qux"]))
```

returns:

```
['BAR', 'BAZ', 'FOO', 'QUX'] # what expected
```

Is first idiom wrong? Could You explain more about this.

==

```
my_list = [3, -4, -2, 5, 1, 7]
sorted(my_list, key=lambda x: abs(x))
```

returns:

```
[1, -2, 3, -4, 5, 7]
```

```
list(filter(lambda x: x>0, my_list)
```

returns:

```
[3, 5, 1, 7]
```

```
list(map(lambda x: abs(x), my_list)
```

returns:

```
[3, 4, 2, 5, 1, 7]
```

One can call other functions (with/without arguments) from inside a lambda function.

```
def foo(msg):
    print(msg)

greet = lambda x = "hello world": foo(x)
greet()
```

prints:

```
hello world
```

This is useful because `lambda` may contain only one expression and by using a subsidiary function one can run multiple statements.

NOTE

Bear in mind that PEP-8 (the official Python style guide) does not recommend assigning lambdas to variables (as shown in the first two examples):

Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier.

Yes:

```
def f(x): return 2*x
```

No:

```
f = lambda x: 2*x
```

The first form means that the name of the resulting function object is specifically f instead of the generic <lambda> . This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression).

Defining a function with optional mutable arguments

There is a problem when using **optional arguments** with a **mutable default type** (described in Defining a function with optional arguments), which can potentially lead to unexpected behaviour.

Explanation

This problem arises because a function's default arguments are initialised **once** , at the point when the function is *defined* , and **not** (like many other languages) when the function is *called* . The default values are stored inside the function object's __defaults__ member variable.

```
def f(a, b=42, c=[]):
    pass

print(f.__defaults__)
# Out: (42, [])
```

For **immutable** types (see Argument passing and mutability) this is not a problem because there is no way to mutate the variable; it can only ever be reassigned, leaving the original value unchanged. Hence, subsequent are guaranteed to have the same default value. However, for a **mutable** type, the original value can mutate, by making calls to its various member functions. Therefore, successive calls to the function are not guaranteed to have the initial default value.

```
def append(elem, to=[]):
    to.append(elem)      # This call to append() mutates the default variable "to"
    return to

append(1)
# Out: [1]

append(2)  # Appends it to the internally stored list
# Out: [1, 2]

append(3, [])  # Using a new created list gives the expected result
# Out: [3]

# Calling it again without argument will append to the internally stored list again
append(4)
# Out: [1, 2, 4]
```

Note: Some IDEs like PyCharm will issue a warning when a mutable type is specified as a default attribute.

Solution

If you want to ensure that the default argument is always the one you specify in the function definition, then the solution is to **always** use an immutable type as your default argument.

A common idiom to achieve this when a mutable type is needed as the default, is to use None (immutable) as the default argument and then assign the actual default value to the argument variable if it is equal to None .

```
def append(elem, to=None):
    if to is None:
        to = []

    to.append(elem)
    return to
```

Argument passing and mutability

First, some terminology:

- **Argument (Actual Parameter):** The actual variable being passed to a function.
- **Parameter (Formal Parameter):** The receiving variable that is used in a function.

In Python, arguments are passed by **assignment** (as opposed to other languages, where arguments could be passed by value/reference/pointer).

- Mutating a parameter will mutate the argument (if the argument's type is mutable).
- Reassigning the parameter won't reassign the argument.

In Python, we don't really assign values to variables.

- Instead, we think of variables as **names** (identifiers, labels, tags) which are **bound** (assigned, attached) to objects.

```
def foo(x):        # The argument (y) is assigned to the parameter (x).
    x[0] = 9       # This mutates the list labelled by both x and y.
    x = [1, 2, 3]  # x is now labelling a different list (y is unaffected).
    x[2] = 8       # This mutates x's list, not y's list.

y = [4, 5, 6]      # y is the argument, x is the parameter.
foo(y)             # Pretend that we wrote "x = y", then go to line 1.
y
# Out: [9, 5, 6]
```

- **Immutable:** Integers, strings, tuples, and so on. All operations make copies.
- **Mutable:** Lists, dictionaries, sets, and so on. Operations may or may not mutate.

```
x = [3, 1, 9]
y = x
x.append(5)    # Mutates the list labelled by both x and y.
x.sort()       # Mutates the list labelled by both x and y (in-place sorting).
x = x + [4]    # Does not mutate the list (makes a copy for x only, not y).
z = x
x += [6]       # Mutates the list labelled by both x and z (uses the extend function).
x = sorted(x)  # Does not mutate the list (makes a copy for x only).
x
# Out: [1, 3, 4, 5, 6, 9]
y
# Out: [1, 3, 5, 9]
z
# Out: [1, 3, 5, 9, 4, 6]
```

Closure

Closures in Python are created by function calls. Here, the call to makeInc creates a binding for x that is referenced inside the function inc . Each call to makeInc creates a new instance of this function, but each instance has a link to a different binding of x .

```
def makeInc(x):
  def inc(y):
     # x is "attached" in the definition of inc
     return y + x

  return inc

incOne = makeInc(1)
incFive = makeInc(5)

incOne (5) # returns 6
incFive(5) # returns 10
```

Notice that while in a regular closure the enclosed function fully inherits all variables from its enclosing environment, in this construct the enclosed function has only read access to the inherited variables but cannot make assignments to them

```
def makeInc(x):
  def inc(y):
     # incrementing x is not allowed
     x += y
     return x

  return inc

incOne = makeInc(1)
incOne(5) # UnboundLocalError: local variable 'x' referenced before assignment
```

Python 3 offers the nonlocal statement (Nonlocal Variables) for realizing a full closure with nested functions.

Python 3.x ≥ 3.0

```
def makeInc(x):
  def inc(y):
     nonlocal x
     # now assigning a value to x is allowed
     x += y
     return x

  return inc

incOne = makeInc(1)
incOne(5) # returns 6
```

Returning values from functions

Functions can `return` a value that you can use directly:

```
def give_me_five():
    return 5

print(give_me_five())  # Print the returned value
# Out: 5
```

or save the value for later use:

```
num = give_me_five()
print(num)             # Print the saved returned value
# Out: 5
```

or use the value for any operations:

```
print(give_me_five() + 10)
# Out: 15
```

If `return` is encountered in the function the function will be exited immediately and subsequent operations will not be evaluated:

```
def give_me_another_five():
    return 5
    print('This statement will not be printed. Ever.')

print(give_me_another_five())
# Out: 5
```

You can also `return` multiple values (in the form of a tuple):

```
def give_me_two_fives():
    return 5, 5  # Returns two 5

first, second = give_me_two_fives()
print(first)
# Out: 5
print(second)
# Out: 5
```

A function with *no* `return` statement implicitly returns `None` . Similarly a function with a `return` statement, but no return value or variable returns `None` .

Defining a function with arguments

Arguments are defined in parentheses after the function name:

```
def divide(dividend, divisor):  # The names of the function and its arguments
    # The arguments are available by name in the body of the function
    print(dividend / divisor)
```

The function name and its list of arguments are called the *signature* of the function. Each named argument is effectively a local variable of the function.

When calling the function, give values for the arguments by listing them in order

```
divide(10, 2)
# output: 5
```

or specify them in any order using the names from the function definition:

```
divide(divisor=2, dividend=10)
# output: 5
```

Forcing the use of named parameters

All parameters specified after the first asterisk in the function signature are keyword-only.

```
def f(*a, b):
    pass

f(1, 2, 3)
# TypeError: f() missing 1 required keyword-only argument: 'b'
```

In Python 3 it's possible to put a single asterisk in the function signature to ensure that the remaining arguments may only be passed using keyword arguments.

```
def f(a, b, *, c):
    pass

f(1, 2, 3)
# TypeError: f() takes 2 positional arguments but 3 were given
f(1, 2, c=3)
# No error
```

Nested functions

Functions in python are first-class objects. They can be defined in any scope

```
def fibonacci(n):
    def step(a,b):
        return b, a+b
    a, b = 0, 1
    for i in range(n):
        a, b = step(a, b)
    return a
```

Functions capture their enclosing scope can be passed around like any other sort of object

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
add5 = make_adder(5)
add6 = make_adder(6)
add5(10)
#Out: 15
add6(10)
#Out: 16

def repeatedly_apply(func, n, x):
    for i in range(n):
        x = func(x)
    return x

repeatedly_apply(add5, 5, 1)
#Out: 26
```

Recursion limit

There is a limit to the depth of possible recursion, which depends on the Python implementation. When the limit is reached, a RuntimeError exception is raised:

```
def cursing(depth):
  try:
    cursing(depth + 1) # actually, re-cursing
  except RuntimeError as RE:
    print('I recursed {} times!'.format(depth))

cursing(0)
# Out: I recursed 1083 times!
```

It is possible to change the recursion depth limit by using sys.setrecursionlimit(limit) and check this limit by sys.getrecursionlimit() .

```
sys.setrecursionlimit(2000)
cursing(0)
# Out: I recursed 1997 times!
```

From Python 3.5, the exception is a RecursionError , which is derived from RuntimeError .

Recursive functions

A recursive function is a function that calls itself in its definition. For example the mathematical function, factorial, defined by factorial(n) = n*(n-1)*(n-2)*...*3*2*1 . can be programmed as

```
def factorial(n):
    #n here should be an integer
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

the outputs here are:

```
factorial(0)
#out 1
factorial(1)
#out 1
factorial(2)
#out 2
factorial(3)
#out 6
```

as expected. Notice that this function is recursive because the second return factorial(n-1) , where the function calls itself in its definition.

Some recursive functions can be implemented using lambda , the factorial function using lambda would be something like this:

```
factorial = lambda n: 1 if n == 0 else n*factorial(n-1)
```

The function outputs the same as above.

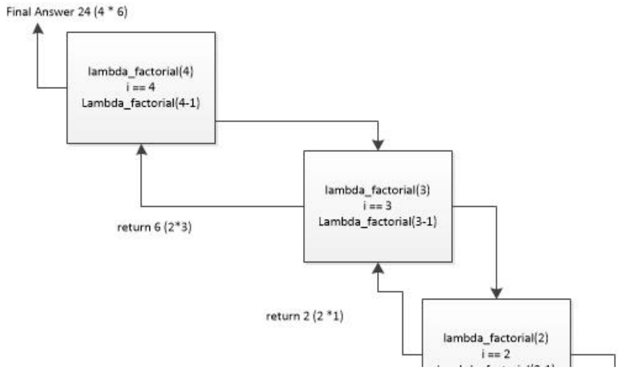
Recursive Lambda using assigned variable

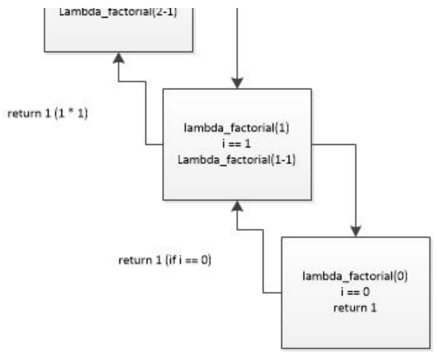
One method for creating recursive lambda functions involves assigning the function to a variable and then referencing that variable within the function itself. A common example of this is the recursive calculation of the factorial of a number - such as shown in the following code:

```
lambda_factorial = lambda i:1 if i==0 else i*lambda_factorial(i-1)
print(lambda_factorial(4)) # 4 * 3 * 2 * 1 = 12 * 2 = 24
```

Description of code

The lambda function, through its variable assignment, is passed a value (4) which it evaluates and returns 1 if it is 0 or else it returns the current value (i) * another calculation by the lambda function of the value - 1 (i-1). This continues until the passed value is decremented to 0 (return 1). A process which can be visualized as:





Defining a function with multiple arguments

One can give a function as many arguments as one wants, the only fixed rules are that each argument name must be unique and that optional arguments must be after the not-optional ones:

```
def func(value1, value2, optionalvalue=10):
    return '{0} {1} {2}'.format(value1, value2, optionalvalue1)
```

When calling the function you can either give each keyword without the name but then the order matters:

```
print(func(1, 'a', 100))
# Out: 1 a 100

print(func('abc', 14))
# abc 14 10
```

Or combine giving the arguments with name and without. Then the ones with name must follow those without but the order of the ones with name doesn't matter:

```
print(func('This', optionalvalue='StackOverflow Documentation', value2='is'))
# Out: This is StackOverflow Documentation
```

Iterable and dictionary unpacking

Functions allow you to specify these types of parameters: positional, named, variable positional, Keyword args (kwargs). Here is a clear and concise use of each type.

```
def unpacking(a, b, c=45, d=60, *args, **kwargs):
    print(a, b, c, d, args, kwargs)

>>> unpacking(1, 2)
1 2 45 60 () {}
>>> unpacking(1, 2, 3, 4)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, c=3)
1 2 3 4 () {}


>>> pair = (3,)
>>> unpacking(1, 2, *pair, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *pair)
1 2 3 4 () {}
>>> unpacking(1, 2, *pair, c=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, c=3, *pair)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'

>>> args_list = [3]
>>> unpacking(1, 2, *args_list, d=4)
1 2 3 4 () {}
>>> unpacking(1, 2, d=4, *args_list)
1 2 3 4 () {}
>>> unpacking(1, 2, c=3, *args_list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unpacking() got multiple values for argument 'c'
>>> unpacking(1, 2, *args_list, c=3)
```

Syntax

```
def function_name ( arg1, ... argN, *args, kw1, kw2=default, ..., **kwargs ): statements
```

```
lambda arg1, ... argN, *args, kw1, kw2=default, ..., **kwargs : expression
```

Parameters

Parameter	Details
arg1 , ..., *argN*	Regular arguments
**args*	Unnamed positional arguments
kw1 , ..., *kwN*	Keyword-only arguments
***kwargs*	The rest of keyword arguments

Remarks

5 key things you can do with functions are elegantly summarized (with examples) in this blog post , which later introduces concepts of closures and decorators:

Assign functions to variables

Define functions inside other functions

Functions can be passed as parameters to other functions

Functions can return other functions

Inner functions have access to the enclosing scope