Python Bootcamp & Masterclass

functions

gknxt

A **function** is a block of code which only runs when it is called

```
def <function_name>(<parameter(s)>):
    <statement(s)>
```

def	The keyword that informs Python that a function is being defined
function_name	A valid Python identifier that names the function
parameter(s)	(optional) A comma-separated list of parameters as input to the function
statement(s)	A block of valid one or more Python statements

```
def hello_world():          # function definition
    print('Hello World')    # function body
hello_world()               # calling the function
```

```
Hello World
```

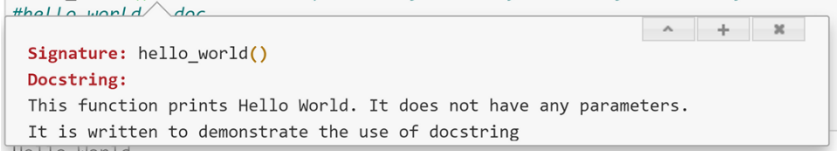
A function is a set of instructions that can be reused/executed repeatedly or that, because of their complexity, are better self-contained in a sub-program and called when needed. It is a piece of code written to carry out a specified task by taking zero or more inputs. When the task is carried out, the function will return one or more values, with **None** being the default. There are three types of functions:

1. built-in functions (e.g. `print()`, `type()`, `len()` etc.)
2. user-defined functions (functions defined by users using `def`)
3. anonymous functions (also called lambda expressions or lambda functions)

docstring

A docstring short for (**doc**umentation **string**) is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the `__doc__` special attribute of that object. Conventions for writing good docstrings are given in PEP 257 (https://www.python.org/dev/peps/pep-0257/)

```
def hello_world():
    """ This function prints Hello World. It does not have any parameters.
        It is written to demonstrate the use of docstring
    """
    print('Hello World')

hello_world()            # press shift+tab anywhere in function definition to see the signature and docstring
#hello_world.__doc__
```




```
Hello World
Help on function hello_world in module __main__:

hello_world()
    This function prints Hello World. It does not have any parameters.
    It is written to demonstrate the use of docstring
```

gk nxt

return

1. Every Python function should return an object. If a user-defined function does not have a `return` statement, Python will implicitly returns **None**

2. The `return` statement immediately terminates the function and passes execution control back to the caller. If a function does not have a return statement, then function terminates after executing the last executable statement. There can be multiple return statements.

3. The `return` statement provides a mechanism by which the function can pass data back to the caller. If multiple comma-separated expressions are specified in a return statement, then they're packed and returned as a tuple

```python
def hello_world():
    return (1, 2, 3, 'Hello World')  # explicit return
hello_world()
type(hello_world())
print(hello_world())
```

```
(1, 2, 3, 'Hello World')

tuple

(1, 2, 3, 'Hello World')
```

```python
def hello_world():
    print('Hello World')             # implicit return

type(hello_world())
print(hello_world())
```

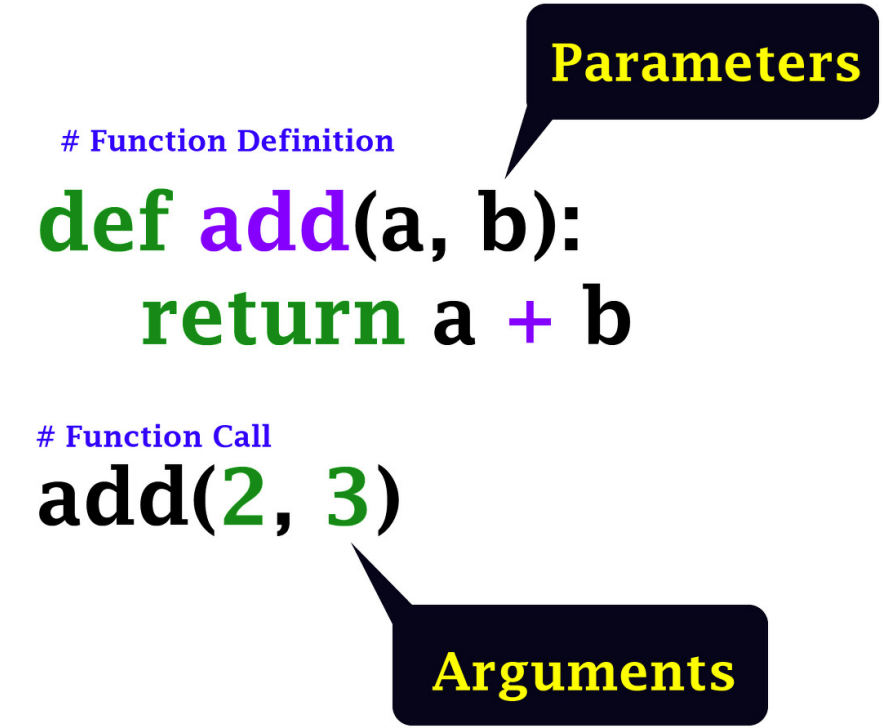
```
Hello World

NoneType

Hello World
None
```

Parameters & Arguments

A parameter is the variable defined inside the parentheses in the function definition.

An argument is the value that is sent to the function when it is called.

The simplest way to pass parameters to a function is by position. In the function definition, the parameters are specified in a particular order and arguments used in the function calling are matched to the function's parameters based on their order.



Positional Arguments

1. With positional arguments, the arguments in the call and the parameters in the definition must agree not only in **order** but in **number** as well.

2. A bare slash (/) in the parameter list of a function definition designate the parameters to the left of the slash (/) must be specified positionally.

Default Parameters

1. Function parameters can have default values, which can be declared by assigning a default value in the function definition.

2. Any number of parameters can be given default values. Parameters with default values must be defined as the last ones in the parameter list.

3. Default parameters are also called optional parameters as they can be omitted when the function is called if the default value is preferred.

4. In Python, default parameter values are defined only once when the function is defined (that is, when the def statement is executed). The default value **will not** be re-defined each time the function is called.

indefinite positional arguments

1. Prefixing the final parameter name of the function with an asterisk (*) causes all excess non-keyword arguments in a call of a function to be collected together and assigned as a tuple to the given parameter.
2. In the definition of a function, the parameter specification ***args** indicates tuple packing. In each call to the function, the arguments are packed into a tuple that the function can refer to by the name **args**. (Any name can be used, but the name args is practically a standard.)

Keyword Arguments

1. Arguments can also be passed into a function by using the name of the parameter rather than its position. Though the order is irrelevant when arguments are passed using keywords, the number of arguments and parameters must still match.

2. When positional and keyword arguments are both present, all the positional arguments must come first.

3. An asterisk (*) in the parameter list of a function definition designate the parameters to the left of the asterisk (*) and to the right of the slash (/), if any, must be specified keyword only.

indefinite keyword arguments

1. Prefixing the final parameter name of the function with a double asterisk (**) causes all excess keyword arguments in a call of a function to be collected together and assigned as a dictionary to the given parameter.
2. In the definition of a function, the parameter specification **kwargs indicates dictionary packing. In each call to the function, the arguments are packed into a dictionary that the function can refer to by the name kwargs. (Any name can be used, but the name kwargs is practically a standard.)
3. The general rule for using mixed argument-passing is that positional arguments come first, then keyword arguments, followed by the indefinite positional arguments and last of all the indefinite keyword arguments

```python
def f2(**kwargs):
    print(kwargs)
    print(type(kwargs))
    for key, val in kwargs.items():
        print(key, '->', val)
f2(a=1, b=2, c=3)
```

```
{'a': 1, 'b': 2, 'c': 3}
<class 'dict'>
a -> 1
b -> 2
c -> 3
```

```python
def concat(**words):
    result = ""
    for arg in words.values():   # words.keys() will get keys
        result += arg
    return result

print(concat(a="Programming ", b="in ", c="Python ", d="is ", e="fun "))
```

```
Programming in Python is fun
```

by value vs by reference

1. In some other programming languages, there are two common paradigms for passing an argument to a function: Pass-by-value and Pass-by-reference. Python follows a different paradigm.

2. Passing an immutable object to a function is like pass-by-value of other languages. The function can't modify the object in the calling environment.

3. Passing a mutable object is somewhat - but not exactly - like pass-by-reference of other languages. The function can't reassign the object wholesale, but it can change items in place within the object, and these changes will be reflected in the calling environment.

```
def f2(my_list=[]):
    my_list.append('###')
    return my_list
```

```
f2([1,2,3])
```

```
[1, 2, 3, '###']
```

```
f2(['a', 'b', 'c'])
```

```
['a', 'b', 'c', '###']
```

```
f2()
f2()
f2()
```

```
['###']
```

```
['###', '###']
```

```
['###', '###', '###']
```

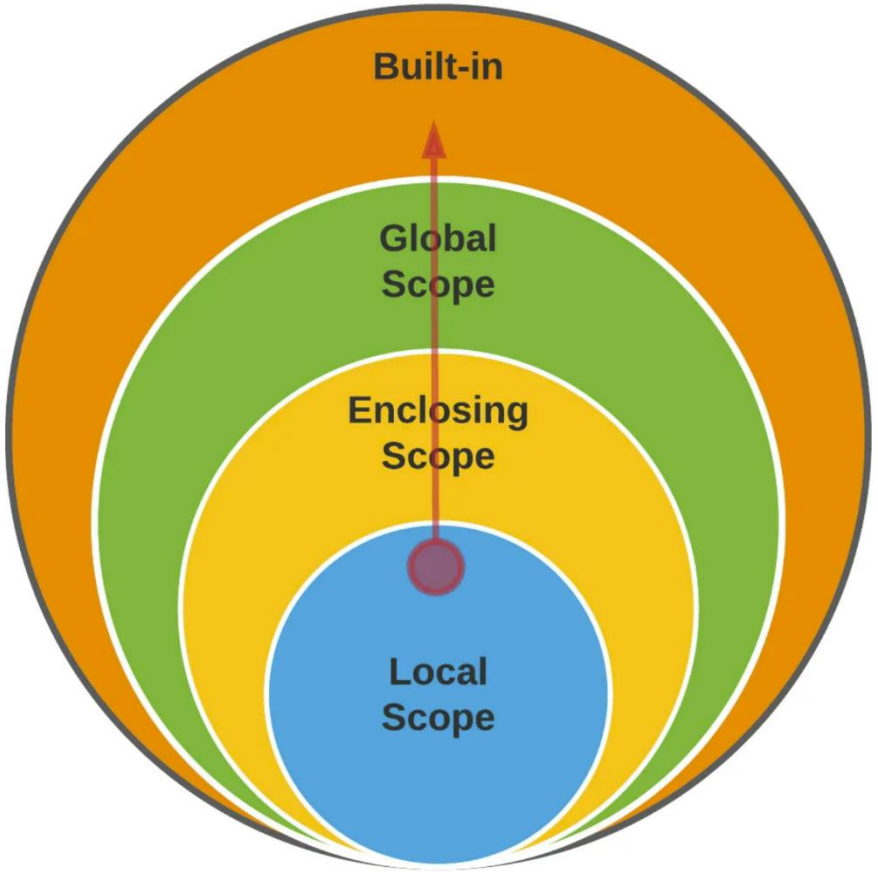
namespace

Namespace in Python is a mapping from identifiers to objects. Scopes are determined by namespaces, which associate identifiers with objects and are implemented as dictionaries. All namespaces are independent of one another. So, the same identifier can appear in multiple namespaces. When a block of code is executed in Python, it has three primary namespaces:

1. local
2. global
3. built-in

For a module, a command executed in an interactive session, or a script running from a file, the global and local namespaces are the same.

When an identifier is encountered during execution, Python first looks in the local namespace for it if the local namespace exists at that point. If the identifier isn't found, Python looks in the enclosing namespace for it if the enclosing namespace exists at that point. If the identifier isn't found, the global namespace is looked in next. If the identifier still hasn't been found, the built-in namespace is checked. If it doesn't exist there, a **NameError** exception occurs.



- Each function and method has a local namespace that associates local identifiers with objects. The local namespace exists from the moment the function or method is called until it terminates and is accessible only to that function or method. Identifiers in the local namespace are in scope from the point at which you define them until the function or method terminates.

- Let's say a main program calls a function `f()`. Python immediately creates a new namespace for `f()`. Let's say `f()` calls `g()`. Python immediately creates a separate namespace for `g()` The namespace created for `g()` is the local namespace, and the namespace created for `f()` is the enclosing namespace.

- The global namespace contains any names defined at the level of the main program. Python creates the global namespace when the main program body starts, and it remains in existence until the interpreter terminates.

- Each module has a global namespace that associates a module's global identifiers (such as global variables, function names and class names) with objects. Python creates a module's global namespace when it loads the module. A module's global namespace exists and its identifiers are in scope to the code within that module until the program (or interactive session) terminates.

- Python creates a global namespace for any module that your program loads with the import statement.

- The built-in namespace associate identifiers for Python's built-in functions and types with objects that define those functions and types. Python creates the built-in namespace when the interpreter starts executing. The built-in namespace's identifiers remain in scope for all code until the program (or interactive session) terminates. The built-in namespace contains the names of all of Python's built-in objects. These are available at all times when Python is running.

- Python allows you to define nested functions inside other functions or methods. When an identifier accessed inside a nested function, Python searches the nested function's local namespace first, then the enclosing function's namespace, then the global namespace and finally the built-in namespace. This is sometimes referred to as the **LEGB** (local, enclosing, global, built-in) rule.

```
z = 'I am z. I am in global scope'       # line 1
def scope_test():                        # line 2
    y = 'I am y. I am in local scope'    # line 3
    print(y)                             # line 4
    print(z)                             # line 5
scope_test()                             # line 6
pass                                     # line 7
```

Assume that Python session just started. It creates built-in namespace.

line 1

Python creates a global namespace and variable z is added to the global namespace

```
z = 'I am z. I am in global scope'       # line 1
def scope_test():                        # line 2
    y = 'I am y. I am in local scope'    # line 3
    print(y)                             # line 4
    print(z)                             # line 5
scope_test()                             # line 6
pass                                     # line 7
```

line 2

function identifier scope_test is added to the global namespace

```
z = 'I am z. I am in global scope'       # line 1
def scope_test():                        # line 2
    y = 'I am y. I am in local scope'    # line 3
    print(y)                             # line 4
    print(z)                             # line 5
scope_test()                             # line 6
pass                                     # line 7
```

line 6

As the execution is not in any function call, only session's global namespace and the built-in namespace are currently accessible. Python first searches the session's global namespace, which contains scope_test. So scope_test is in scope and Python uses the corresponding object to call scope_test

```
z = 'I am z. I am in global scope'       # line 1
def scope_test():                        # line 2
    y = 'I am y. I am in local scope'    # line 3
    print(y)                             # line 4
    print(z)                             # line 5
scope_test()                             # line 6
pass                                     # line 7
```

line 3

As scope_test begins executing, Python creates the function's local namespace.

As function scope_test defined a variable y, Python adds y to the function's local namespace. The variable y is now in scope until the function finishes executing.

```
z = 'I am z. I am in global scope'       # line 1
def scope_test():                        # line 2
    y = 'I am y. I am in local scope'    # line 3
    print(y)                             # line 4
    print(z)                             # line 5
scope_test()                             # line 6
pass                                     # line 7
```

line 4

scope_test calls the built-in function print, passing y as the argument. To execute this call, Python must resolve the identifiers y and print. The identifier y is defined in the local namespace, so it's in scope and Python will use the corresponding object (the string 'I am y. I am in local scope') as print's argument. To call the function, Python must find print's corresponding object. First, it looks in the local namespace, which does not define print. Next, it looks in the session's global namespace, which does not define print. Finally, it looks in the built-in namespace, which does define print. So, print is in scope and Python uses the corresponding object to call print.

```
z = 'I am z. I am in global scope'       # line 1
def scope_test():                        # line 2
    y = 'I am y. I am in local scope'    # line 3
    print(y)                             # line 4
    print(z)                             # line 5
scope_test()                             # line 6
pass                                     # line 7
```

line 5

scope_test calls the built-in function print again with the argument z, which is not defined in the local namespace. So, Python looks in the global namespace. The argument z is defined in the global namespace, so z is in scope and Python will use the corresponding object (the string 'I am z. I am in global scope') as print's argument. Again, Python finds the identifier print in the built-in namespace and uses the corresponding object to call print.

```
z = 'I am z. I am in global scope'       # line 1
def scope_test():                        # line 2
    y = 'I am y. I am in local scope'    # line 3
    print(y)                             # line 4
    print(z)                             # line 5
scope_test()                             # line 6
pass                                     # line 7
```

line 7

After executing line 5, function terminates as it is the last statement and there is no return statement. The function's local namespace no longer exists, meaning the local variable y is now undefined.

Online Resources

For best python resources, please visit:

gknxt.com/python/

gknxt

Python
Bootcamp
& Masterclass

Thank You

for your Rating & Review

gknxt