

Namespaces and Scope in Python

by [John Sturtz](#) 5 Comments

[basics](#) [python](#)

Mark as Completed [?](#)

Table of Contents

- [Namespaces in Python](#)
 - [The Built-In Namespace](#)
 - [The Global Namespace](#)
 - [The Local and Enclosing Namespaces](#)
- [Variable Scope](#)
- [Python Namespace Dictionaries](#)
 - [The `globals\(\)` function](#)
 - [The `locals\(\)` function](#)
- [Modify Variables Out of Scope](#)
 - [The `global` Declaration](#)
 - [The `nonlocal` Declaration](#)
 - [Best Practices](#)
- [Conclusion](#)

[? Tweet](#) [? Share](#) [? Email](#)

This tutorial covers Python **namespaces**, the structures used to organize the symbolic names assigned to objects in a Python program.

The previous tutorials in this series have emphasized the importance of [objects](#) in Python. Objects are everywhere! Virtually everything that your Python program creates or acts on is an object.

An **assignment statement** creates a **symbolic name** that you can use to reference an object. The statement `x = 'foo'` creates a symbolic name `x` that refers to the [string](#) object `'foo'`.

In a program of any complexity, you’ll create hundreds or thousands of such names, each pointing to a specific object. How does Python keep track of all these names so that they don’t interfere with one another?

In this tutorial, you'll learn:

- How Python organizes symbolic names and objects in **namespaces**
- When Python creates a new namespace
- How namespaces are implemented
- How **variable scope** determines symbolic name visibility

y**y**

Namespaces in Python

A namespace is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a [dictionary](#) in which the keys are the object names and the values are the objects themselves. Each key-value pair maps a name to its corresponding object.

As Tim Peters suggests, namespaces aren't just great. They're *honking* great, and Python uses them extensively. In a Python program, there are four types of namespaces:

1. Built-In
2. Global
3. Enclosing
4. Local

These have differing lifetimes. As Python executes a program, it creates namespaces as necessary and deletes them when they're no longer needed. Typically, many namespaces will exist at any given time.

The Built-In Namespace

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. You can list the objects in the built-in namespace with the following command:

Python

>>>

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
 'BaseException', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
 'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError',
 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError',
 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
 'InterruptedError', 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
 'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError',
 'OverflowError', 'PendingDeprecationWarning', 'PermissionError',
 'ProcessLookupError', 'RecursionError', 'ReferenceError', 'ResourceWarning',
 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIteration',
 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
 'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError',
 'Warning', 'ZeroDivisionError', '_', '__build_class__', '__debug__',
 '__doc__', '__import__', '__loader__', '__name__', '__package__',
 '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',
 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate',
 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

You'll see some objects here that you may recognize from previous tutorials—for example, the [StopIteration](#) exception, [built-in functions](#) like `max()` and `len()`, and object types like `int` and `str`.

The Python interpreter creates the built-in namespace when it starts up. This namespace remains in existence until the interpreter terminates.

The Global 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.

Strictly speaking, this may not be the only global namespace that exists. The interpreter also creates a global namespace for any **module** that your program loads with the [import](#) statement. For further reading on main functions and modules in Python, see these resources:

- [Defining Main Functions in Python](#)
- [Python Modules and Packages—An Introduction Course:](#)
- [Python Modules and Packages](#)

You'll explore modules in more detail in a future tutorial in this series. For the moment, when you see the term *global namespace*, think of the one belonging to the main program.

The Local and Enclosing Namespaces

As you learned in the previous tutorial on [functions](#), the interpreter creates a new namespace whenever a function executes. That namespace is local to the function and remains in existence until the function terminates.

Functions don't exist independently from one another only at the level of the main program. You can also [define one function inside another](#):

```
Python >>>
1 >>> def f():
2 ...     print('Start f()')
3 ...
4 ...     def g():
5 ...         print('Start g()')
6 ...         print('End g()')
7 ...         return
8 ...
9 ...     g()
10 ...
11 ...     print('End f()')
12 ...     return
13 ...
14
15 >>> f()
16 Start f()
17 Start g()
18 End g()
19 End f()
```

In this example, function `g()` is defined within the body of `f()`. Here's what's happening in this code:

- **Lines 1 to 12** define `f()`, the **enclosing** function.
- **Lines 4 to 7** define `g()`, the **enclosed** function.
- On **line 15**, the main program calls `f()`.
- On **line 9**, `f()` calls `g()`.

When the main program calls `f()`, Python creates a new namespace for `f()`. Similarly, when `f()` calls `g()`, `g()` gets its own separate namespace. The namespace created for `g()` is the **local namespace**, and the namespace created for `f()` is the **enclosing namespace**.

Each of these namespaces remains in existence until its respective function terminates. Python might not immediately reclaim the memory allocated for those namespaces when their functions terminate, but all references to the objects they contain cease to be valid.

Variable Scope

The existence of multiple, distinct namespaces means several different instances of a particular name can exist simultaneously while a Python program runs. As long as each instance is in a different namespace, they're all maintained separately and won't interfere with one another.

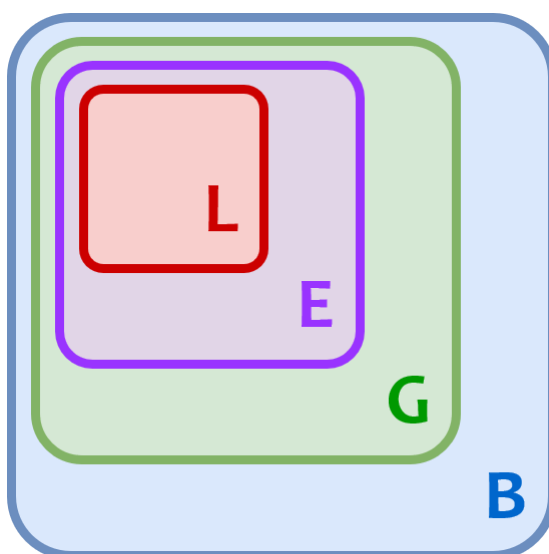
But that raises a question: Suppose you refer to the name `x` in your code, and `x` exists in several namespaces. How does Python know which one you mean?

The answer lies in the concept of **scope**. The [scope](#) of a name is the region of a program in which that name has meaning. The interpreter determines this at runtime based on where the name definition occurs and where in the code the name is referenced.

To return to the above question, if your code refers to the name `x`, then Python searches for `x` in the following namespaces in the order shown:

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**: If `x` isn't in the local scope but appears in a function that resides inside another function, then the interpreter searches in the enclosing function's scope.
3. **Global**: If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
4. **Built-in**: If it can't find `x` anywhere else, then the interpreter tries the built-in scope.

This is the **LEGB rule** as it's commonly called in Python literature (although the term doesn't actually appear in the [Python documentation](#)). The interpreter searches for a name from the inside out, looking in the local, enclosing, global, and finally the built-in scope:



If the interpreter doesn't find the name in any of these locations, then Python raises a [NameError exception](#).

Examples

Several examples of the LEGB rule appear below. In each case, the innermost enclosed function `g()` attempts to display the value of a variable named `x` to the console. Notice how each example prints a different value for `x` depending on its scope.

Example 1: Single Definition

In the first example, `x` is defined in only one location. It's outside both `f()` and `g()`, so it resides in the global scope:

Python

>>>

```
1 >>> x = 'global'
2
3 >>> def f():
4 ...
5 ...     def g():
6 ...         print(x)
7 ...
8 ...     g()
9 ...
10
11 >>> f()
12 global
```

The `print()` statement on **line 6** can refer to only one possible `x`. It displays the `x` object defined in the global namespace, which is the string `'global'`.

Example 2: Double Definition

In the next example, the definition of `x` appears in two places, one outside `f()` and one inside `f()` but outside `g()`:

Python

>>>

```
1 >>> x = 'global'
2
3 >>> def f():
4 ...     x = 'enclosing'
5 ...
6 ...     def g():
7 ...         print(x)
8 ...
9 ...     g()
10 ...
11
12 >>> f()
13 enclosing
```

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'`.

Example 3: Triple Definition

Next is a situation in which `x` is defined here, there, and everywhere. One definition is outside `f()`, another one is inside `f()` but outside `g()`, and a third is inside `g()`:

Python

>>>

```
1 >>> 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()
14 local
```

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'`.

Example 4: No Definition

Last, we have a case in which `g()` tries to print the value of `x`, but `x` isn't defined anywhere. That won't work at all:

```
Python                                                                 >>>
1  >>> def f():
2  ...
3  ...     def g():
4  ...         print(x)
5  ...
6  ...     g()
7  ...
8
9  >>> f()
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12   File "<stdin>", line 6, in f
13   File "<stdin>", line 4, in g
14   NameError: name 'x' is not defined
```

This time, Python doesn't find `x` in any of the namespaces, so the `print()` statement on **line 4** generates a `NameError` exception.

Python Namespace Dictionaries

Earlier in this tutorial, when [namespaces were first introduced](#), you were encouraged to think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves. In fact, for global and local namespaces, that's precisely what they are! Python really does implement these namespaces as dictionaries.

Python provides built-in functions called `globals()` and `locals()` that allow you to access global and local namespace dictionaries.

The `globals()` function

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. Here's an example of what it looks like when the main program starts:

```
Python                                                                 >>>
>>> type(globals())
<class 'dict'>

>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>}
```

As you can see, the interpreter has put several entries in `globals()` already. Depending on your Python version and operating system, it may look a little different in your environment. But it should be similar.

Now watch what happens when you define a variable in the global scope:

Python

>>>

```
>>> x = 'foo'

>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
 '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
 'x': 'foo'}
```

After the assignment statement `x = 'foo'`, a new item appears in the global namespace dictionary. The dictionary key is the object's name, `x`, and the dictionary value is the object's value, `'foo'`.

You would typically access this object in the usual way, by referring to its symbolic name, `x`. But you can also access it indirectly through the global namespace dictionary:

Python

>>>

```
1 >>> x
2 'foo'
3 >>> globals()['x']
4 'foo'
5
6 >>> x is globals()['x']
7 True
```

The [is comparison](#) on **line 6** confirms that these are in fact the same object.

You can create and modify entries in the global namespace using the `globals()` function as well:

Python

>>>

```
1 >>> globals()['y'] = 100
2
3 >>> globals()
4 {'__name__': '__main__', '__doc__': None, '__package__': None,
5  '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
6  '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
7  'x': 'foo', 'y': 100}
8
9 >>> y
10 100
11
12 >>> globals()['y'] = 3.14159
13
14 >>> y
15 3.14159
```

The statement on **line 1** has the same effect as the assignment statement `y = 100`. The statement on **line 12** is equivalent to `y = 3.14159`.

It's a little off the beaten path to create and modify objects in the global scope this way when simple assignment statements will do. But it works, and it illustrates the concept nicely.

The `locals()` function

Python also provides a corresponding built-in function called `locals()`. It's similar to `globals()` but accesses objects in the local namespace instead:

Python

>>>

```
>>> def f(x, y):
...     s = 'foo'
...     print(locals())
...
>>> f(10, 0.5)
{'s': 'foo', 'y': 0.5, 'x': 10}
```


When called within `f()`, `locals()` returns a dictionary representing the function's local namespace. Notice that, in addition to the locally defined variable `s`, the local namespace includes the function parameters `x` and `y` since these are local to `f()` as well.

If you call `locals()` outside a function in the main program, then it behaves the same as `globals()`.

Deep Dive: A Subtle Difference Between `globals()` and `locals()`

There's one small difference between `globals()` and `locals()` that's useful to know about.

`globals()` returns an actual reference to the dictionary that contains the global namespace. That means if you call `globals()`, save the return value, and subsequently define additional variables, then those new variables will show up in the dictionary that the saved return value points to:

Python

>>>

```
1 >>> g = globals()
2 >>> g
3 {'__name__': '__main__', '__doc__': None, '__package__': None,
4  '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
5  '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
6  'g': {...}}
7
8 >>> x = 'foo'
9 >>> y = 29
10 >>> g
11 {'__name__': '__main__', '__doc__': None, '__package__': None,
12  '__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__': None,
13  '__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
14  'g': {...}, 'x': 'foo', 'y': 29}
```

Here, `g` is a reference to the global namespace dictionary. After the assignment statements on **lines 8 and 9**, `x` and `y` appear in the dictionary that `g` points to.

`locals()`, on the other hand, returns a dictionary that is a current copy of the local namespace, not a reference to it. Further additions to the local namespace won't affect a previous return value from `locals()` until you call it again. Also, you can't modify objects in the actual local namespace using the return value from `locals()`:

Python

>>>

```
1 >>> def f():
2 ...     s = 'foo'
3 ...     loc = locals()
4 ...     print(loc)
5 ...
6 ...     x = 20
7 ...     print(loc)
8 ...
9 ...     loc['s'] = 'bar'
10 ...    print(s)
11 ...
12
13 >>> f()
14 {'s': 'foo'}
15 {'s': 'foo'}
16 foo
```

In this example, `loc` points to the return value from `locals()`, which is a copy of the local namespace. The statement `x = 20` on **line 6** adds `x` to the local namespace but *not* to the copy that `loc` points to. Similarly, the statement on **line 9** modifies the value for key `'s'` in the copy that `loc` points to, but this has no effect on the value of `s` in the actual local namespace.

It's a subtle difference, but it could cause you trouble if you don't remember it.

Modify Variables Out of Scope

Earlier in this series, in the tutorial on [user-defined Python functions](#), you learned that argument passing in Python is a bit like [pass-by-value](#) and a bit like [pass-by-reference](#). Sometimes a function can modify its argument in the calling environment by making changes to the corresponding parameter, and sometimes it can't:

- An **immutable** argument can never be modified by a function.
- A **mutable** argument can't be redefined wholesale, but it can be modified in place.

Note: For more information on modifying function arguments, see [Pass-By-Value vs Pass-By-Reference in Pascal and Pass-By-Value vs Pass-By-Reference in Python](#).

A similar situation exists when a function tries to modify a variable outside its local scope. A function can't modify an [immutable](#) object outside its local scope at all:

Python

>>>

```
1 >>> x = 20
2 >>> def f():
3 ...     x = 40
4 ...     print(x)
5 ...
6
7 >>> f()
8 40
9 >>> x
10 20
```

When `f()` executes the assignment `x = 40` on **line 3**, it creates a new local [reference](#) to an integer object whose value is 40. At that point, `f()` loses the reference to the object named `x` in the global namespace. So the assignment statement doesn't affect the global object.

Note that when `f()` executes `print(x)` on **line 4**, it displays 40, the value of its own local `x`. But after `f()` terminates, `x` in the global scope is still 20.

A function can modify an object of mutable type that's outside its local scope if it modifies the object in place:

Python

>>>

```
>>> my_list = ['foo', 'bar', 'baz']
>>> def f():
...     my_list[1] = 'quux'
...
>>> f()
>>> my_list
['foo', 'quux', 'baz']
```

In this case, `my_list` is a list, and lists are mutable. `f()` can make changes inside `my_list` even though it's outside the local scope.

But if `f()` tries to reassign `my_list` entirely, then it will create a new local object and won't modify the global `my_list`:

Python

>>>

```
>>> my_list = ['foo', 'bar', 'baz']
>>> def f():
...     my_list = ['qux', 'quux']
...
>>> f()
>>> my_list
['foo', 'bar', 'baz']
```

This is similar to what happens when `f()` tries to modify a mutable function argument.

The global Declaration

What if you really do need to modify a value in the global scope from within `f()`? This is possible in Python using the `global` declaration:

Python

>>>

```
>>> x = 20
>>> def f():
...     global x
...     x = 40
...     print(x)
...

>>> f()
40

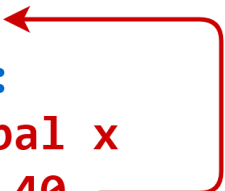
>>> x
40
```

The `global x` statement indicates that while `f()` executes, references to the name `x` will refer to the `x` that is in the global namespace. That means the assignment `x = 40` doesn't create a new reference. It assigns a new value to `x` in the global scope instead:

```
>>> x = 20
>>> def f():
...     global x
...     x = 40
...     print(x)
...

>>> f()
40

>>> x
40
```



The global Declaration

As you've already seen, `globals()` returns a reference to the global namespace dictionary. If you wanted to, instead of using a `global` statement, you could accomplish the same thing using `globals()`:

Python

>>>

```
>>> x = 20
>>> def f():
...     globals()['x'] = 40
...     print(x)
...

>>> f()
40

>>> x
40
```

There isn't much reason to do it this way since the `global` declaration arguably makes the intent clearer. But it does provide another illustration of how `globals()` works.

If the name specified in the `global` declaration doesn't exist in the global scope when the function starts, then a combination of the `global` statement and an assignment will create it:

Python

>>>

```
1 >>> y
2 Traceback (most recent call last):
3   File "<pyshell#79>", line 1, in <module>
4     y
5 NameError: name 'y' is not defined
6
7 >>> def g():
8     ...     global y
9     ...     y = 20
10    ...
11
12 >>> g()
13 >>> y
14 20
```

In this case, there's no object named `y` in the global scope when `g()` starts, but `g()` creates one with the `global y` statement on **line 8**.

You can also specify several comma-separated names in a single `global` declaration:

Python

>>>

```
1 >>> x, y, z = 10, 20, 30
2
3 >>> def f():
4     ...     global x, y, z
5     ...
```

Here, `x`, `y`, and `z` are all declared to refer to objects in the global scope by the single `global` statement on **line 4**.

A name specified in a `global` declaration can't appear in the function prior to the `global` statement:

Python

>>>

```
1 >>> def f():
2     ...     print(x)
3     ...     global x
4     ...
5   File "<stdin>", line 3
6 SyntaxError: name 'x' is used prior to global declaration
```

The intent of the `global x` statement on **line 3** is to make references to `x` refer to an object in the global scope. But the `print()` statement on **line 2** refers to `x` prior to the `global` declaration. This raises a [SyntaxError](#) exception.

The nonlocal Declaration

A similar situation exists with nested function definitions. The `global` declaration allows a function to access and modify an object in the global scope. What if an enclosed function needs to modify an object in the enclosing scope? Consider this example:

In this case, the first definition of `x` is in the enclosing scope, not the global scope. Just as `g()` can't directly modify a variable in the global scope, neither can it modify `x` in the enclosing function's scope. Following the assignment `x = 40` on **line 5**, `x` in the enclosing scope remains 20.

The [global keyword](#) isn't a solution for this situation:

```
Python >>>
1 >>> def f():
2 ...     x = 20
3 ...
4 ...     def g():
5 ...         x = 40
6 ...
7 ...     g()
8 ...     print(x)
9 ...
10
11 >>> f()
12 20
```

```
Python >>>
>>> def f():
...     x = 20
...
...     def g():
...         global x
...         x = 40
...
...     g()
...     print(x)
...
>>> f()
20
```

Since `x` is in the enclosing function's scope, not the global scope, the `global` keyword doesn't work here. After `g()` terminates, `x` in the enclosing scope remains 20.

In fact, in this example, the `global x` statement not only fails to provide access to `x` in the enclosing scope, but it also creates an object called `x` in the global scope whose value is 40:

```
Python >>>
>>> def f():
...     x = 20
...
...     def g():
...         global x
...         x = 40
...
...     g()
...     print(x)
...
>>> f()
20
>>> x
40
```

To modify `x` in the enclosing scope from inside `g()`, you need the analogous keyword [nonlocal](#). Names specified after the `nonlocal` keyword refer to variables in the nearest enclosing scope:

After the `nonlocal x` statement on **line 5**, when `g()` refers to `x`, it refers to the `x` in the nearest enclosing scope, whose definition is in `f()` on **line 2**:

Python

```
1 >>> def f():
2 ...     x = 20
3 ...
4 ...     def g():
5 ...         nonlocal x
6 ...         x = 40
7 ...
8 ...     g()
9 ...     print(x)
10 ...
11
12 >>> f()
13 40
```

```
>>> def f():
...     x = 20
...
...     def g():
...         nonlocal x
...         x = 40
...
...     g()
...     print(x)
...

>>> f()
40
```

The nonlocal Declaration

The `print()` statement at the end of `f()` on **line 9** confirms that the call to `g()` has changed the value of `x` in the enclosing scope to 40.

Best Practices

Even though Python provides the `global` and `nonlocal` keywords, it’s not always advisable to use them.

When a function modifies data outside the local scope, either with the `global` or `nonlocal` keyword or by directly modifying a mutable type in place, it’s a kind of [side effect](#) similar to when a function modifies one of its arguments. Widespread modification of [global variables](#) is generally considered unwise, not only in Python but also in other programming languages.

As with many things, this is somewhat a matter of style and preference. There are times when judicious use of global variable modification can reduce program complexity.

In Python, using the `global` keyword at least makes it explicit that the function is modifying a global variable. In many languages, a function can modify a global variable just by assignment, without announcing it in any way. This can make it very difficult to track down where global data is being modified.

All in all, modifying variables outside the local scope usually isn’t necessary. There’s almost always a better way, usually with function return values.

Conclusion

Virtually everything that a Python program uses or acts on is an object. Even a short program will create many different objects. In a more complex program, they'll probably number in the thousands. Python has to keep track of all these objects and their names, and it does so with **namespaces**.

In this tutorial, you learned:

- What the different **namespaces** are in Python
- When Python creates a new namespace
- What structure Python uses to implement namespaces How
- namespaces define **scope** in a Python program

Many programming techniques take advantage of the fact that every function in Python has its own namespace. In the next two tutorials in this series, you'll explore two of these techniques: **functional programming** and **recursion**.

