Python Modules and Packages – An Introduction

This article explores Python **modules** and Python **packages**, two mechanisms that facilitate **modular programming**.

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**. Individual modules can then be cobbled together like building blocks to create a larger application.

There are several advantages to **modularizing** code in a large application:

- **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.
- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. (You may even be able to make changes to a module without having any knowledge of the application outside that module.) This makes it more viable for a team of many programmers to work collaboratively on a large application.
- **Reusability:** Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to duplicate code.
- **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program. (One of the tenets in the Zen of Python is *Namespaces are one honking great idea—let's do more of those!*)

Functions, **modules** and **packages** are all constructs in Python that promote code modularization.

Python Modules: Overview

There are actually three different ways to define a **module** in Python:

1. A module can be written in Python itself.
2. A module can be written in **C** and loaded dynamically at run-time, like the `re` (**regular expression**) module.
3. A **built-in** module is intrinsically contained in the interpreter, like the `itertools` module.

A module's contents are accessed the same way in all three cases: with the `import` statement.

Here, the focus will mostly be on modules that are written in Python. The cool thing about modules written in Python is that they are exceedingly straightforward to build. All you need to do is create a file that contains legitimate Python code and then give the file a name with a `.py` extension. That's it! No special syntax or voodoo is necessary.

For example, suppose you have created a file called `mod.py` containing the following:

mod.py

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

Several objects are defined in `mod.py`:

- `s` (a string)
- `a` (a list)
- `foo()` (a function)
- `Foo` (a class)

Assuming `mod.py` is in an appropriate location, which you will learn more about shortly, these objects can be accessed by **importing** the module as follows:

```
>>> import mod
>>> print(mod.s)
If Comrade Napoleon says it, it must be right.
>>> mod.a
[100, 200, 300]
>>> mod.foo(['quux', 'corge', 'grault'])
arg = ['quux', 'corge', 'grault']
>>> x = mod.Foo()
>>> x
<mod.Foo object at 0x03C181F0>
```

The Module Search Path

Continuing with the above example, let's take a look at what happens when Python executes the statement:

```
import mod
```

When the interpreter executes the above `import` statement, it searches for `mod.py` in a list of directories assembled from the following sources:

- The directory from which the input script was run or the **current directory** if the interpreter is being run interactively
- The list of directories contained in the `PYTHONPATH` environment variable, if it is set. (The format for `PYTHONPATH` is OS-dependent but should mimic the `PATH` environment variable.)
- An installation-dependent list of directories configured at the time Python is installed

The resulting search path is accessible in the Python variable `sys.path`, which is obtained from a module named `sys`:

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\john\\Documents\\Python\\doc',
'C:\\Python36\\Lib\\idlelib',
'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',
'C:\\Python36', 'C:\\Python36\\lib\\site-packages']
```

Note: The exact contents of `sys.path` are installation-dependent. The above will almost certainly look slightly different on your computer.

Thus, to ensure your module is found, you need to do one of the following:

- Put `mod.py` in the directory where the input script is located or the **current directory**, if interactive
- Modify the `PYTHONPATH` environment variable to contain the directory where `mod.py` is located before starting the interpreter
    - **Or:** Put `mod.py` in one of the directories already contained in the `PYTHONPATH` variable
- Put `mod.py` in one of the installation-dependent directories, which you may or may not have write-access to, depending on the OS

There is actually one additional option: you can put the module file in any directory of your choice and then modify `sys.path` at run-time so that it contains that directory. For example, in this case, you could put `mod.py` in directory `C:\Users\john` and then issue the following statements:

```
>>> sys.path.append(r'C:\Users\john')
>>> sys.path
['', 'C:\\Users\\john\\Documents\\Python\\doc',
'C:\\Python36\\Lib\\idlelib',
'C:\\Python36\\python36.zip', 'C:\\Python36\\DLLs', 'C:\\Python36\\lib',
'C:\\Python36', 'C:\\Python36\\lib\\site-packages', 'C:\\Users\\john']
>>> import mod
```

Once a module has been imported, you can determine the location where it was found with the module's `__file__` attribute:

```
>>> import mod
>>> mod.__file__
'C:\\Users\\john\\mod.py'

>>> import re
>>> re.__file__
'C:\\Python36\\lib\\re.py'
```

The directory portion of `__file__` should be one of the directories in `sys.path`.

The `import` Statement

Module contents are made available to the caller with the `import` statement. The `import` statement takes many different forms, shown below.

`import <module_name>`

The simplest form is the one already shown above:

```
import <module_name>
```

Note that this *does not* make the module contents *directly* accessible to the caller. Each module has its own **private symbol table**, which serves as the global symbol table for all objects defined *in the module*. Thus, a module creates a separate **namespace**, as already noted.

The statement `import <module_name>` only places `<module_name>` in the caller's symbol table. The *objects* that are defined in the module *remain in the module's private symbol table*.

From the caller, objects in the module are only accessible when prefixed with `<module_name>` via **dot notation**, as illustrated below.

After the following `import` statement, `mod` is placed into the local symbol table. Thus, `mod` has meaning in the caller's local context:

```
>>> import mod
>>> mod
<module 'mod' from 'C:\\Users\\john\\Documents\\Python\\doc\\mod.py'>
```

But `s` and `foo` remain in the module's private symbol table and are not meaningful in the local context:

```
>>> s
NameError: name 's' is not defined
>>> foo('quux')
NameError: name 'foo' is not defined
```

To be accessed in the local context, names of objects defined in the module must be prefixed by `mod`:

```
>>> mod.s
'If Comrade Napoleon says it, it must be right.'
>>> mod.foo('quux')
arg = quux
```

Several comma-separated modules may be specified in a single `import` statement:

```
import <module_name>[, <module_name> ...]
```

`from <module_name> import <name(s)>`

An alternate form of the `import` statement allows individual objects from the module to be imported *directly into the caller's symbol table*:

```
from <module_name> import <name(s)>
```

Following execution of the above statement, `<name(s)>` can be referenced in the caller's environment without the `<module_name>` prefix:

```
>>> from mod import s, foo
>>> s
```

```
'If Comrade Napoleon says it, it must be right.'
>>> foo('quux')
arg = quux

>>> from mod import Foo
>>> x = Foo()
>>> x
<mod.Foo object at 0x02E3AD50>
```

Because this form of `import` places the object names directly into the caller's symbol table, any objects that already exist with the same name will be *overwritten*:

```
>>> a = ['foo', 'bar', 'baz']
>>> a
['foo', 'bar', 'baz']

>>> from mod import a
>>> a
[100, 200, 300]
```

It is even possible to indiscriminately `import` everything from a module at one fell swoop:

```
from <module_name> import *
```

This will place the names of *all* objects from `<module_name>` into the local symbol table, with the exception of any that begin with the underscore (_) character.

For example:

```
>>> from mod import *
>>> s
'If Comrade Napoleon says it, it must be right.'
>>> a
[100, 200, 300]
>>> foo
<function foo at 0x03B449C0>
>>> Foo
<class 'mod.Foo'>
```

This isn't necessarily recommended in large-scale production code. It's a bit dangerous because you are entering names into the local symbol table *en masse*. Unless you know them all well and can be confident there won't be a conflict, you have a decent chance of overwriting an existing name inadvertently. However, this syntax is quite handy when you are just mucking around with the interactive interpreter, for testing or discovery purposes, because it quickly gives you access to everything a module has to offer without a lot of typing.

`from <module_name> import <name> as <alt_name>`

It is also possible to `import` individual objects but enter them into the local symbol table with alternate names:

```
from <module_name> import <name> as <alt_name>[, <name> as <alt_name> …]
```

This makes it possible to place names directly into the local symbol table but avoid conflicts with previously existing names:

```
>>> s = 'foo'
>>> a = ['foo', 'bar', 'baz']

>>> from mod import s as string, a as alist
>>> s
'foo'
>>> string
'If Comrade Napoleon says it, it must be right.'
>>> a
['foo', 'bar', 'baz']
>>> alist
[100, 200, 300]
```

`import <module_name> as <alt_name>`

You can also import an entire module under an alternate name:

```
import <module_name> as <alt_name>
>>> import mod as my_module
>>> my_module.a
[100, 200, 300]
>>> my_module.foo('qux')
arg = qux
```

Module contents can be imported from within a function definition. In that case, the `import` does not occur until the function is *called*:

```
>>> def bar():
...     from mod import foo
...     foo('corge')
...

>>> bar()
arg = corge
```

However, **Python 3** does not allow the indiscriminate `import *` syntax from within a function:

```
>>> def bar():
...     from mod import *
...
SyntaxError: import * only allowed at module level
```

Lastly, a `try` statement with an `except ImportError` clause can be used to guard against unsuccessful `import` attempts:

```
>>> try:
...     # Non-existent module
...     import baz
... except ImportError:
...     print('Module not found')
...

Module not found
>>> try:
...     # Existing module, but non-existent object
...     from mod import baz
... except ImportError:
...     print('Object not found in module')
```

```
...
Object not found in module
```

The `dir()` Function

The built-in function `dir()` returns a list of defined names in a namespace. Without arguments, it produces an alphabetically sorted list of names in the current **local symbol table**:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> qux = [1, 2, 3, 4, 5]
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'qux']

>>> class Bar():
...     pass
...
>>> x = Bar()
>>> dir()
['Bar', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__', 'qux', 'x']
```

Note how the first call to `dir()` above lists several names that are automatically defined and already in the namespace when the interpreter starts. As new names are defined (`qux`, `Bar`, `x`), they appear on subsequent invocations of `dir()`.

This can be useful for identifying what exactly has been added to the namespace by an import statement:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> import mod
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'mod']
>>> mod.s
'If Comrade Napoleon says it, it must be right.'
>>> mod.foo([1, 2, 3])
arg = [1, 2, 3]

>>> from mod import a, Foo
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__', 'a', 'mod']
>>> a
[100, 200, 300]
>>> x = Foo()
>>> x
<mod.Foo object at 0x002EAD50>
```

```
>>> from mod import s as string
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__', 'a', 'mod', 'string', 'x']
>>> string
'If Comrade Napoleon says it, it must be right.'
```

When given an argument that is the name of a module, `dir()` lists the names defined in the module:

```
>>> import mod
>>> dir(mod)
['Foo', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'a', 'foo', 's']
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
>>> from mod import *
>>> dir()
['Foo', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__', 'a', 'foo', 's']
```

Executing a Module as a Script

Any `.py` file that contains a **module** is essentially also a Python **script**, and there isn't any reason it can't be executed like one.

Here again is `mod.py` as it was defined above:

mod.py

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass
```

This can be run as a script:

```
C:\Users\john\Documents>python mod.py
C:\Users\john\Documents>
```

There are no errors, so it apparently worked. Granted, it's not very interesting. As it is written, it only *defines* objects. It doesn't *do* anything with them, and it doesn't generate any output.

Let's modify the above Python module so it does generate some output when run as a script:

mod.py

```
s = "If Comrade Napoleon says it, it must be right."
```

```
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass

print(s)
print(a)
foo('quux')
x = Foo()
print(x)
```

Now it should be a little more interesting:

```
C:\Users\john\Documents>python mod.py
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<__main__.Foo object at 0x02F101D0>
```

Unfortunately, now it also generates output when imported as a module:

```
>>> import mod
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<mod.Foo object at 0x0169AD50>
```

This is probably not what you want. It isn't usual for a module to generate output when it is imported.

Wouldn't it be nice if you could distinguish between when the file is loaded as a module and when it is run as a standalone script?

Ask and ye shall receive.

When a `.py` file is imported as a module, Python sets the special **dunder** variable `__name__` to the name of the module. However, if a file is run as a standalone script, `__name__` is (creatively) set to the string `'__main__'`. Using this fact, you can discern which is the case at run-time and alter behavior accordingly:

mod.py

```
s = "If Comrade Napoleon says it, it must be right."
a = [100, 200, 300]

def foo(arg):
    print(f'arg = {arg}')

class Foo:
    pass

if (__name__ == '__main__'):
    print('Executing as standalone script')
    print(s)
    print(a)
    foo('quux')
    x = Foo()
    print(x)
```

Now, if you run as a script, you get output:

```
C:\Users\john\Documents>python mod.py
Executing as standalone script
If Comrade Napoleon says it, it must be right.
[100, 200, 300]
arg = quux
<__main__.Foo object at 0x03450690>
```

But if you import as a module, you don't:

```
>>> import mod
>>> mod.foo('grault')
arg = grault
```

Modules are often designed with the capability to run as a standalone script for purposes of testing the functionality that is contained within the module. This is referred to as **unit testing**. For example, suppose you have created a module `fact.py` containing a **factorial** function, as follows:

fact.py

```
def fact(n):
    return 1 if n == 1 else n * fact(n-1)

if (__name__ == '__main__'):
    import sys
    if len(sys.argv) > 1:
        print(fact(int(sys.argv[1])))
```

The file can be treated as a module, and the `fact()` function imported:

```
>>> from fact import fact
>>> fact(6)
720
```

But it can also be run as a standalone by passing an integer argument on the command-line for testing:

```
C:\Users\john\Documents>python fact.py 6
720
```

Reloading a Module

For reasons of efficiency, a module is only loaded once per interpreter session. That is fine for function and class definitions, which typically make up the bulk of a module's contents. But a module can contain executable statements as well, usually for initialization. Be aware that these statements will only be executed the *first time* a module is imported.

Consider the following file `mod.py`:

mod.py

```
a = [100, 200, 300]
print('a =', a)
>>> import mod
a = [100, 200, 300]
>>> import mod
>>> import mod

>>> mod.a
[100, 200, 300]
```

The `print()` statement is not executed on subsequent imports. (For that matter, neither is the assignment statement, but as the final display of the value of `mod.a` shows, that doesn't matter. Once the assignment is made, it sticks.)

If you make a change to a module and need to reload it, you need to either restart the interpreter or use a function called `reload()` from module `importlib`:

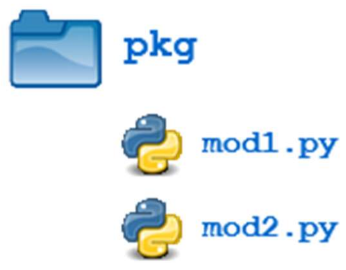
```
>>> import mod
a = [100, 200, 300]

>>> import mod

>>> import importlib
>>> importlib.reload(mod)
a = [100, 200, 300]
<module 'mod' from 'C:\\Users\\john\\Documents\\Python\\doc\\mod.py'>
```

Python Packages

Suppose you have developed a very large application that includes many modules. As the number of modules grows, it becomes difficult to keep track of them all if they are dumped into one location. This is particularly so if they have similar names or functionality. You might wish for a means of grouping and organizing them.

Packages allow for a hierarchical structuring of the module namespace using **dot notation**. In the same way that **modules** help avoid collisions between global variable names, **packages** help avoid collisions between module names.

Creating a **package** is quite straightforward, since it makes use of the operating system's inherent hierarchical file structure. Consider the following arrangement:



Here, there is a directory named `pkg` that contains two modules, `mod1.py` and `mod2.py`. The contents of the modules are:

mod1.py

```
def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

mod2.py

```
def bar():
    print('[mod2] bar()')

class Bar:
    pass
```

Given this structure, if the `pkg` directory resides in a location where it can be found (in one of the directories contained in `sys.path`), you can refer to the two **modules** with **dot notation** (`pkg.mod1`, `pkg.mod2`) and import them with the syntax you are already familiar with:

```
import <module_name>[, <module_name> ...]
>>> import pkg.mod1, pkg.mod2
>>> pkg.mod1.foo()
[mod1] foo()
>>> x = pkg.mod2.Bar()
>>> x
<pkg.mod2.Bar object at 0x033F7290>
from <module_name> import <name(s)>
>>> from pkg.mod1 import foo
>>> foo()
[mod1] foo()
from <module_name> import <name> as <alt_name>
>>> from pkg.mod2 import Bar as Qux
>>> x = Qux()
>>> x
<pkg.mod2.Bar object at 0x036DFFD0>
```

You can import modules with these statements as well:

```
from <package_name> import <modules_name>[, <module_name> ...]
from <package_name> import <module_name> as <alt_name>
>>> from pkg import mod1
>>> mod1.foo()
[mod1] foo()

>>> from pkg import mod2 as quux
>>> quux.bar()
[mod2] bar()
```

You can technically import the package as well:

```
>>> import pkg
>>> pkg
<module 'pkg' (namespace)>
```

But this is of little avail. Though this is, strictly speaking, a syntactically correct Python statement, it doesn't do much of anything useful. In particular, it *does not place* any of the modules in `pkg` into the local namespace:

```
>>> pkg.mod1
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    pkg.mod1
AttributeError: module 'pkg' has no attribute 'mod1'
>>> pkg.mod1.foo()
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    pkg.mod1.foo()
AttributeError: module 'pkg' has no attribute 'mod1'
>>> pkg.mod2.Bar()
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    pkg.mod2.Bar()
AttributeError: module 'pkg' has no attribute 'mod2'
```

To actually import the modules or their contents, you need to use one of the forms shown above.

Package Initialization

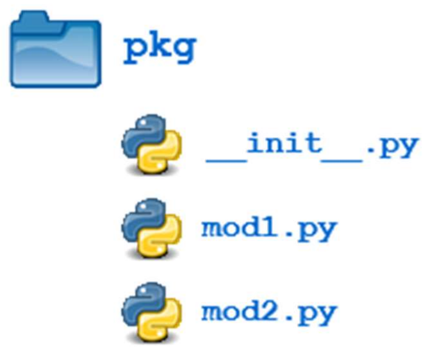
If a file named `__init__.py` is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.

For example, consider the following `__init__.py` file:

__init__.py

```
print(f'Invoking __init__.py for {__name__}')
A = ['quux', 'corge', 'grault']
```

Let's add this file to the `pkg` directory from the above example:



Now when the package is imported, the global list `A` is initialized:

```
>>> import pkg
Invoking __init__.py for pkg
>>> pkg.A
['quux', 'corge', 'grault']
```

A **module** in the package can access the global variable by importing it in turn:

mod1.py

```
def foo():
    from pkg import A
    print('[mod1] foo() / A = ', A)

class Foo:
    pass
>>> from pkg import mod1
Invoking __init__.py for pkg
>>> mod1.foo()
[mod1] foo() / A =  ['quux', 'corge', 'grault']
```

`__init__.py` can also be used to effect automatic importing of modules from a package. For example, earlier you saw that the statement `import pkg` only places the name `pkg` in the caller's local symbol table and doesn't import any modules. But if `__init__.py` in the `pkg` directory contains the following:

__init__.py

```
print(f'Invoking __init__.py for {__name__}')
import pkg.mod1, pkg.mod2
```

then when you execute `import pkg`, modules `mod1` and `mod2` are imported automatically:

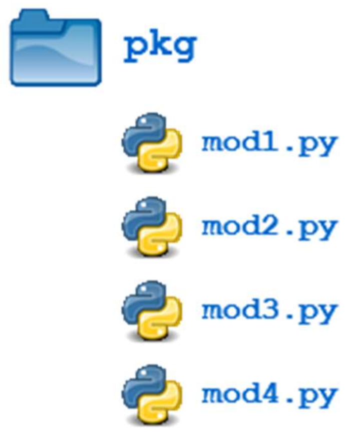
```
>>> import pkg
Invoking __init__.py for pkg
>>> pkg.mod1.foo()
[mod1] foo()
>>> pkg.mod2.bar()
[mod2] bar()
```

Note: Much of the Python documentation states that an `__init__.py` file **must** be present in the package directory when creating a package. This was once true. It used to be that the very presence of `__init__.py` signified to Python that a package was being defined. The file could contain initialization code or even be empty, but it **had** to be present.

Starting with **Python 3.3**, Implicit Namespace Packages were introduced. These allow for the creation of a package without any `__init__.py` file. Of course, it **can** still be present if package initialization is needed. But it is no longer required.

Importing `*` From a Package

For the purposes of the following discussion, the previously defined package is expanded to contain some additional modules:



There are now four modules defined in the `pkg` directory. Their contents are as shown below:

mod1.py

```
def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

mod2.py

```
def bar():
    print('[mod2] bar()')

class Bar:
    pass
```

mod3.py

```
def baz():
    print('[mod3] baz()')

class Baz:
    pass
```

mod4.py

```
def qux():
    print('[mod4] qux()')

class Qux:
    pass
```

(Imaginative, aren't they?)

You have already seen that when `import *` is used for a **module**, *all* objects from the module are imported into the local symbol table, except those whose names begin with an underscore, as always:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> from pkg.mod3 import *

>>> dir()
['Baz', '__annotations__', '__builtins__', '__doc__', '__loader__',
'__name__',
'__package__', '__spec__', 'baz']
>>> baz()
[mod3] baz()
>>> Baz
<class 'pkg.mod3.Baz'>
```

The analogous statement for a **package** is this:

```
from <package_name> import *
```

What does that do?

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> from pkg import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']
```

Hmph. Not much. You might have expected (assuming you had any expectations at all) that Python would dive down into the package directory, find all the modules it could, and import them all. But as you can see, by default that is not what happens.

Instead, Python follows this convention: if the `__init__.py` file in the **package** directory contains a **list** named `__all__`, it is taken to be a list of modules that should be imported when the statement `from <package_name> import *` is encountered.

For the present example, suppose you create an `__init__.py` in the `pkg` directory like this:

pkg/__init__.py

```
__all__ = [
        'mod1',
        'mod2',
        'mod3',
        'mod4'
        ]
```

Now `from pkg import *` imports all four modules:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> from pkg import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'mod1', 'mod2', 'mod3', 'mod4']
```

```
>>> mod2.bar()
[mod2] bar()
>>> mod4.Qux
<class 'pkg.mod4.Qux'>
```

Using `import *` still isn't considered terrific form, any more for **packages** than for **modules**. But this facility at least gives the creator of the package some control over what happens when `import *` is specified. (In fact, it provides the capability to disallow it entirely, simply by declining to define `__all__` at all. As you have seen, the default behavior for packages is to import nothing.)

By the way, `__all__` can be defined in a **module** as well and serves the same purpose: to control what is imported with `import *`. For example, modify `mod1.py` as follows:

pkg/mod1.py

```
__all__ = ['foo']

def foo():
    print('[mod1] foo()')

class Foo:
    pass
```

Now an `import *` statement from `pkg.mod1` will only import what is contained in `__all__`:

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__']

>>> from pkg.mod1 import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'foo']

>>> foo()
[mod1] foo()
>>> Foo
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    Foo
NameError: name 'Foo' is not defined
```

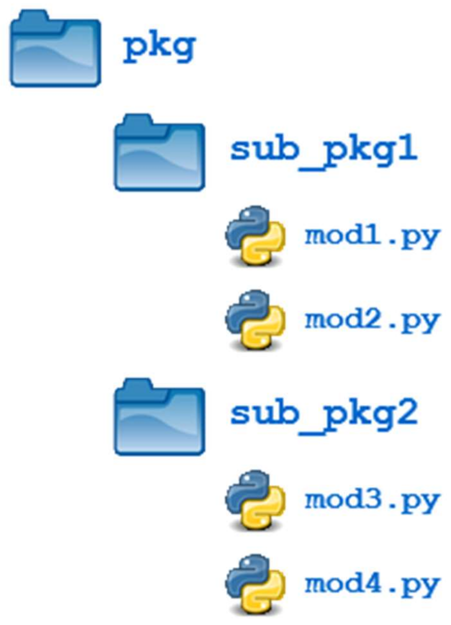
`foo()` (the function) is now defined in the local namespace, but `Foo` (the class) is not, because the latter is not in `__all__`.

In summary, `__all__` is used by both **packages** and **modules** to control what is imported when `import *` is specified. But *the default behavior differs*:

- For a package, when `__all__` is not defined, `import *` does not import anything.
- For a module, when `__all__` is not defined, `import *` imports everything (except—you guessed it—names starting with an underscore).

Subpackages

Packages can contain nested **subpackages** to arbitrary depth. For example, let’s make one more modification to the example **package** directory as follows:



The four modules (`mod1.py`, `mod2.py`, `mod3.py` and `mod4.py`) are defined as previously. But now, instead of being lumped together into the `pkg` directory, they are split out into two **subpackage** directories, `sub_pkg1` and `sub_pkg2`.

Importing still works the same as shown previously. Syntax is similar, but additional **dot notation** is used to separate **package** name from **subpackage** name:

```
>>> import pkg.sub_pkg1.mod1
>>> pkg.sub_pkg1.mod1.foo()
[mod1] foo()

>>> from pkg.sub_pkg1 import mod2
>>> mod2.bar()
[mod2] bar()

>>> from pkg.sub_pkg2.mod3 import baz
>>> baz()
[mod3] baz()

>>> from pkg.sub_pkg2.mod4 import qux as grault
>>> grault()
[mod4] qux()
```

In addition, a module in one **subpackage** can reference objects in a **sibling subpackage** (in the event that the sibling contains some functionality that you need). For example, suppose you want to import and execute function `foo()` (defined in module `mod1`) from within module `mod3`. You can either use an **absolute import**:

pkg/sub__pkg2/mod3.py

```
def baz():
    print('[mod3] baz()')
```

```
class Baz:
    pass

from pkg.sub_pkg1.mod1 import foo
foo()
>>> from pkg.sub_pkg2 import mod3
[mod1] foo()
>>> mod3.foo()
[mod1] foo()
```

Or you can use a **relative import**, where `..` refers to the package one level up. From within `mod3.py`, which is in subpackage `sub_pkg2`,

- `..` evaluates to the parent package (`pkg`), and
- `..sub_pkg1` evaluates to subpackage `sub_pkg1` of the parent package.

pkg/sub__pkg2/mod3.py

```
def baz():
    print('[mod3] baz()')

class Baz:
    pass

from .. import sub_pkg1
print(sub_pkg1)

from ..sub_pkg1.mod1 import foo
foo()
>>> from pkg.sub_pkg2 import mod3
<module 'pkg.sub_pkg1' (namespace)>
[mod1] foo()
```

Conclusion

In this tutorial, you covered the following topics:

- How to create a Python **module**
- Locations where the Python interpreter searches for a module
- How to obtain access to the objects defined in a module with the `import` statement
- How to create a module that is executable as a standalone script
- How to organize modules into **packages** and **subpackages**
- How to control package initialization

Namespaces and Scope in Python

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

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.

Namespaces are one honking great idea—let's do more of those!

— *The Zen of Python, by Tim Peters*

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:

```
>>> 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:

```
>>> def f():
...     print('Start f()')
...
...     def g():
...         print('Start g()')
...         print('End g()')
...         return
...
...     g()
...
...     print('End f()')
...     return
...

>>> f()
Start f()
Start g()
End g()
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.

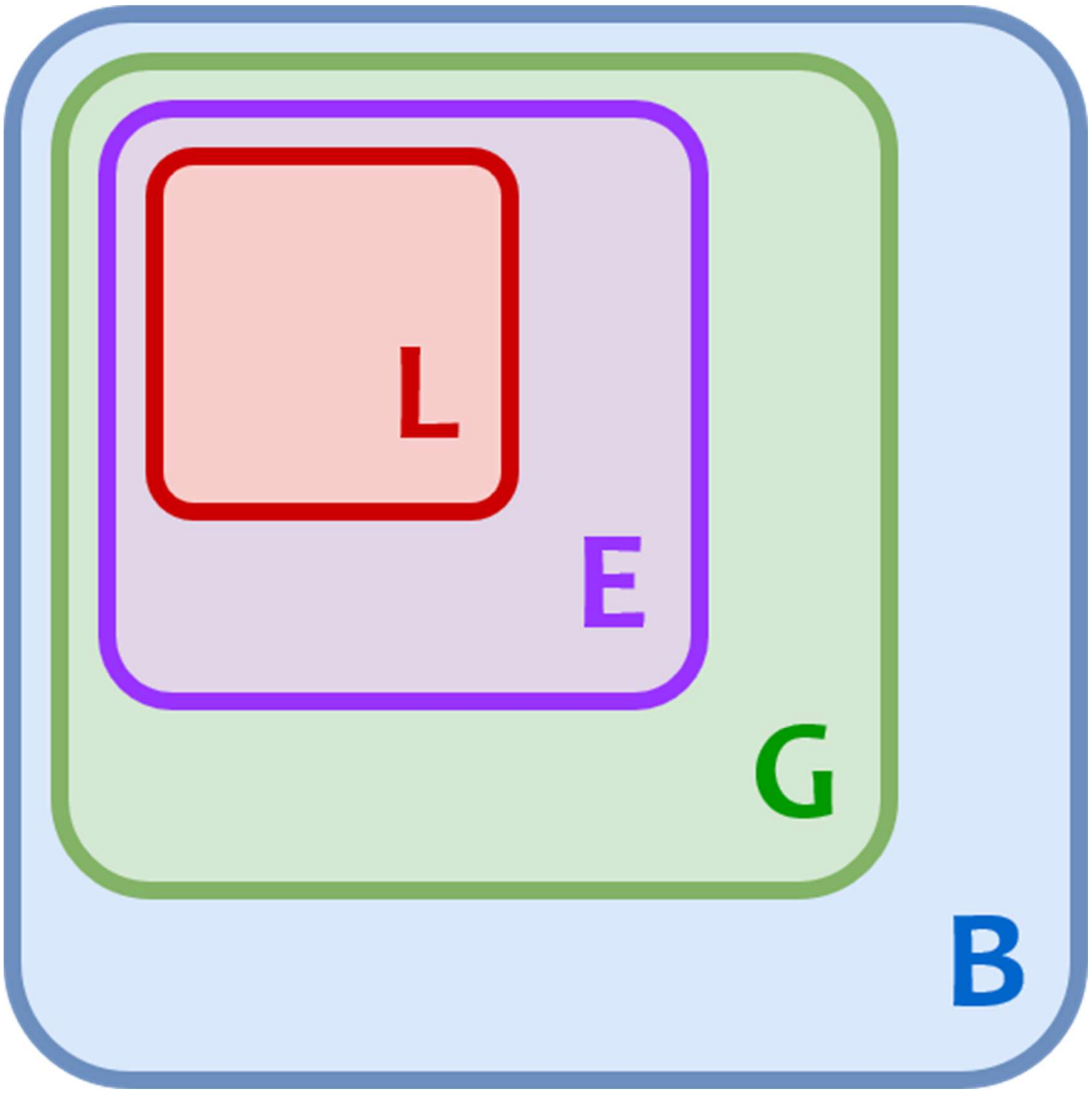
Further Reading: See the Wikipedia page on scope in computer programming for a very thorough discussion of variable scope in programming languages.

If you'd prefer to dive into a video course, then check out Exploring Scopes and Closures in Python or get back to the fundamentals with Python Basics: Scopes.

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 **l**ocal, **e**nclosing, **g**lobal, and finally the **b**uilt-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:

```
>>> x = 'global'

>>> def f():
...
...     def g():
...         print(x)
...
...     g()
```

```
...
>>> f()
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()`:

```
>>> x = 'global'

>>> def f():
...     x = 'enclosing'
...
...     def g():
...         print(x)
...
...     g()
...

>>> f()
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()`:

```
>>> x = 'global'

>>> def f():
...     x = 'enclosing'
...
...     def g():
...         x = 'local'
...         print(x)
...
...     g()
...

>>> f()
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:

```
>>> def f():
...
...     def g():
...         print(x)
...
...     g()
...

>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in f
  File "<stdin>", line 4, in g
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.

Note: The built-in namespace doesn't behave like a dictionary. Python implements it as a module.

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:

```
>>> 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:

```
>>> 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:

```
>>> x
'foo'
>>> globals()['x']
'foo'

>>> x is globals()['x']
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:

```
>>> globals()['y'] = 100

>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__':
None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
'x': 'foo', 'y': 100}

>>> y
100

>>> globals()['y'] = 3.14159

>>> y
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:

```
>>> 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:

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

>>> x = 'foo'
>>> y = 29
>>> g
{'__name__': '__main__', '__doc__': None, '__package__': None,
'__loader__': <class '_frozen_importlib.BuiltinImporter'>, '__spec__':
None,
'__annotations__': {}, '__builtins__': <module 'builtins' (built-in)>,
'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()`:

```
>>> def f():
...     s = 'foo'
...     loc = locals()
...     print(loc)
...
...     x = 20
...     print(loc)
...
...     loc['s'] = 'bar'
...     print(s)
...

>>> f()
{'s': 'foo'}
{'s': 'foo'}
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:

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

>>> f()
40
>>> x
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:

```
>>> 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`:

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

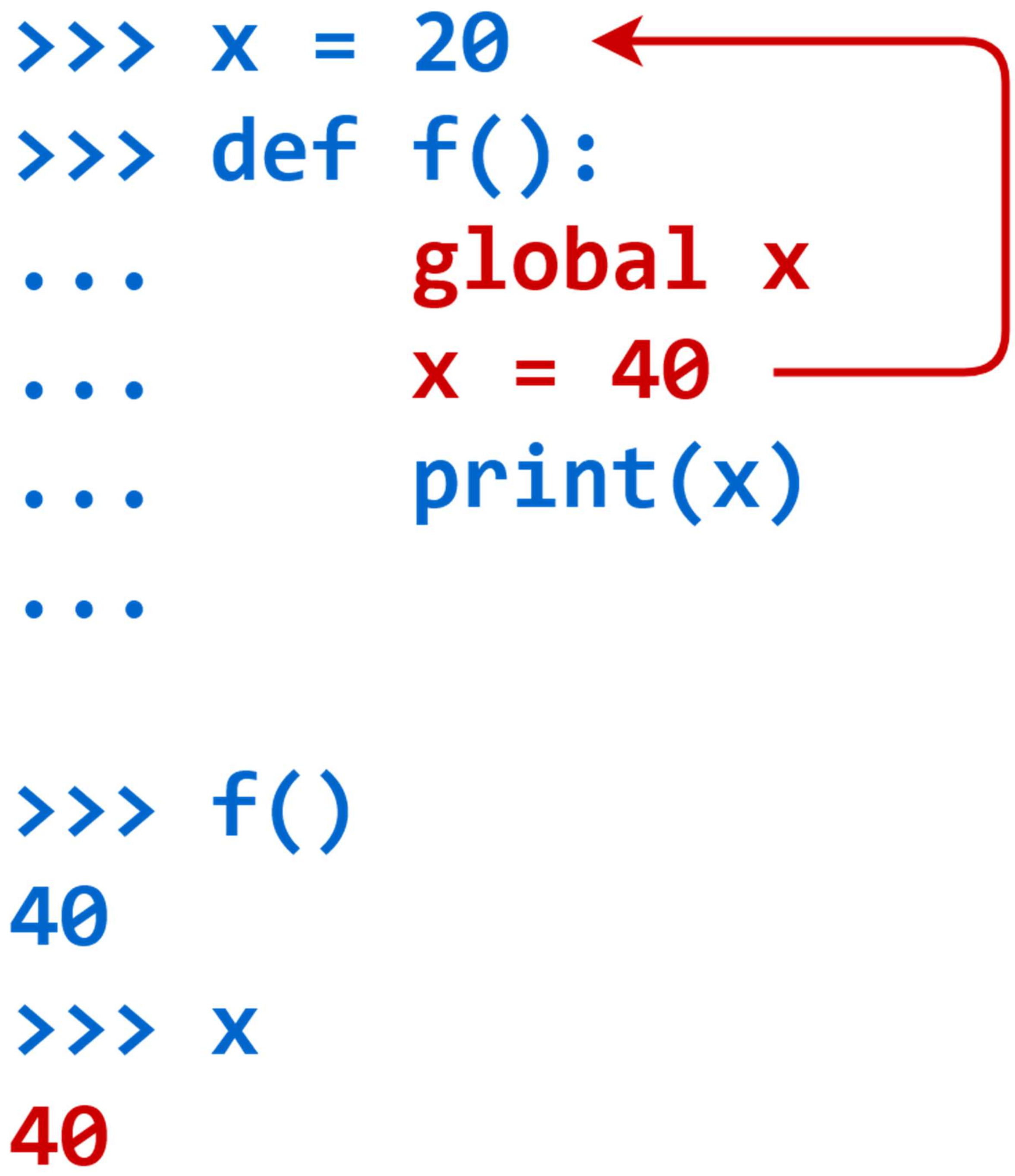
This is similar to what happens when `f()` tries to modify a mutable function argument.

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:

```
>>> 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()`:

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

>>> f()
40
>>> x
```

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:

```
>>> y
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    y
NameError: name 'y' is not defined

>>> def g():
...     global y
...     y = 20
...

>>> g()
>>> y
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:

```
>>> x, y, z = 10, 20, 30

>>> def f():
...     global x, y, z
...
```

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:

```
>>> def f():
...     print(x)
...     global x
...
  File "<stdin>", line 3
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` to 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:

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

>>> f()
20
```

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:

```
>>> 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`:

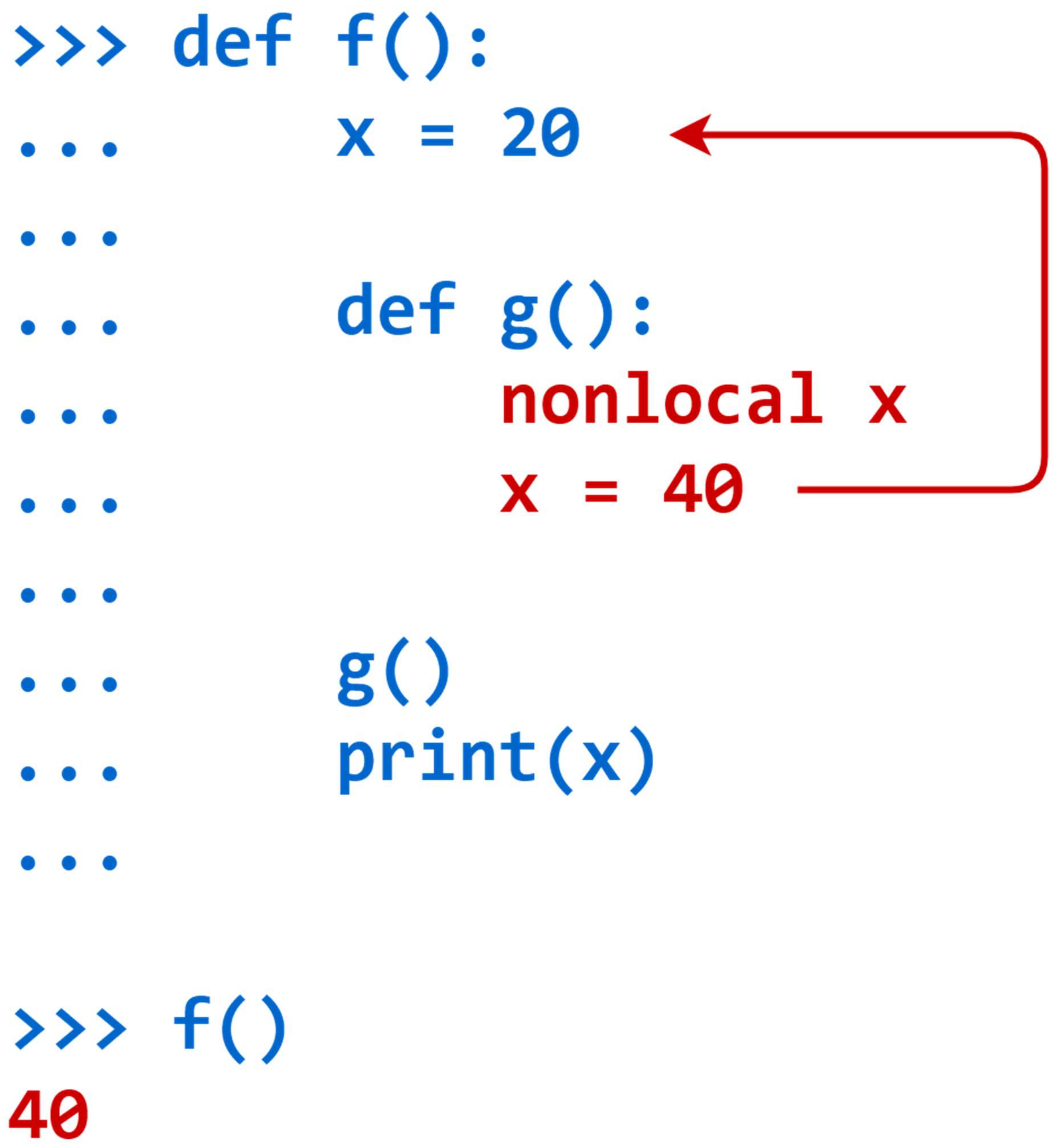
```
>>> def f():
...     x = 20
...
...     def g():
...         global x
...         x = 40
...
...     g()
...     print(x)
...

>>> 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:

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

>>> f()
40
```

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**:



```
>>> 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**.