Published in The Startup

Adrian Mark Perea Follow
Oct 13, 2019 · 6 min read · · Listen

Save

Dictionaries aren't that much different from Namespaces. Source: Pixabay

# Mastering Python Namespaces and Scopes

The official Python documentation defines namespaces and scopes like this:

> *A namespace* is a mapping from names to objects.
>
> *A scope is a* textual region of a Python program where a namespace is directly accessible.

*Say what now*?

Let's put it in more concrete terms. A *namespace* determines which identifiers (e.g. variables, functions, classes) are available for use, and a *scope* defines where — in your written code — a namespace can be accessed. Simple, right?

No? Okay, that's on me. Let's break it down even further.

## For Starters

Whenever you define a variable, Python needs a way to remember two things: the name (identifier) of the variable, and the value you assigned to it. Internally, Python keeps track of all of these definitions by implicitly adding them to a dictionary, mapping the name of each variable you define to its value.

This internal dictionary serves as a lookup table for all your variables. Whenever you try to access a variable, the Python interpreter looks its name up in the dictionary and, if found, returns you its value. If not, it throws a *NameError*.

Let's look at an example:

```python
a = 1 # namespace {a: 1}

# does `a` exist in the namespace? Yes!
# proceed without error
print(a) # => 1

# does `b` exist in the namespace? No.
# Throw a `NameError`
print(b) # => NameError: name `b` is not defined.

b = 16 # namespace {a: 1, b: 16}

# no more problems!
print(b)
```

namespaces.py hosted with ❤ by GitHub view raw

Namespaces Example

In essence, this is simply what a namespace is: an internal dictionary used as a lookup table for names. Simple!

But not so fast; let's go one step further.

## Into the Rabbit Hole

In reality, there are multiple namespaces existing at any given time while a Python program is running, and which namespace you have access to is determined by the *scope* you are currently in. A scope, in essence, is a *textual area* in your program that decides which of these multiple namespaces you have access to.

There are four types of scopes, each more specific than the last. A more specific scope can access the namespace of a less specific scope, but not the other way around. Here are the four types, arranged from most specific to least specific:

1. **The local scope.** The local scope is determined by whether you are in a class/function definition or not. Inside a class/function, the local scope refers to the names defined inside them. Outside a class/function, the local scope is the same as the global scope.

2. **The non-local scope.** A non-local scope is midways between the local scope and the global scope, e.g. the non-local scope of a function defined *inside* another function is the enclosing function itself.

3. **The global scope.** This refers to the scope outside any functions or class definitions. It also known as the module scope.

4. **The built-ins scope.** This scope, as the name suggests, is a scope that is built into Python. While it resides in its own module, any Python program is qualified to call the names defined here without requiring special access.

A visual example can easily clear this up:

```
# scopes.py

# Scope A
a = 1
b = 16

```

```python
def outer():
    # Scope B
    c = 24
    d = 'Hello, World!'

    def inner():
        # Scope C
        e = 'I like'
        f = 'fried chicken'

# (Implicit) Scope D
print('Hello!')
```

scopes.py hosted with ❤ by GitHub view raw

2. **Scope B.** From the perspective of Scope B it is the local scope, but from the perspective of scope C, it is the non-local scope. Scope A has no access to the scope inside Scope B.

3. **Scope C.** From the perspective of Scope C, it is the local scope. Scopes A and B have no access to Scope C.

4. **Scope D.** This is the built-ins scope. All other scopes have access to it.

It is important to know how to distinguish these scopes because the scope you are currently in determines which namespaces are available for you to use.

Python uses the concept of scopes to search for the variable you are trying to access. Whenever you try to access a variable, Python searches in the following order:

1. Local scope

2. Non-local scope

3. Global scope

4. Built-ins scope

> Think of scopes as nested Matryoshka Dolls: Python searches for a name from the innermost doll, and subsequently searches each larger doll until it reaches the outermost one. If it doesn't find after reaching the largest doll, it throws an error. It's Python's way of saying *wait, you have no more dolls!*

Matryoshka Dolls. Source: Pixabay

To illustrate variable access from different scopes, consider the following example:

```python
less_specific = 5

def foo():
    more_specific = 2
    print('Inside foo:', less_specific)

foo()
print(more_specific)
```

variable_access.py hosted with ❤ by GitHub view raw

Which outputs:

```
Inside foo: 5
NameError: 'more_specific' is not defined
```

variable_access_res.py hosted with ❤ by GitHub view raw

The reason behind the error is that `foo()` has access to the global namespace, while the global namespace doesn't have access to the local namespace of the function.

An important thing to remember is that Python searches for names *outwards* from inner scopes, and not the other way around! This means that you can be assured of two things:

- More specific namespaces (e.g. the local namespace inside a function) will never be altered by less specific namespaces (e.g. the global namespace)
- More specific namespaces will always have access to less specific namespaces

## The global and nonlocal keywords

I mentioned in the previous section that more specific namespaces have access to less specific namespaces. Does this mean that functions can alter global variables? Consider the following example:

```
i_am_global = 5

def foo():
    i_am_global = 10
    print(i_am_global)

foo()
print(i_am_global)

# outputs
10
5
```

globals.py hosted with ❤ by GitHub view raw

Modifying Global Variables Locally

The reason behind the result is that while inner scopes *do* have access to outer scopes, they only have *read-only* access to them. The moment that `i_am_global` is altered inside the function, a copy of that variable is created inside the local namespace, thus preserving the value of the global `i_am_global` .

While generally considered bad practice, by prefixing the global variable with the keyword `global` inside the function, a copy of the variable won't be created in the local namespace; that is to say, accessing a variable prefixed with `global` removes the read-only constraint and allows you to alter its value:

```
i_am_global = 5
```

```python
def foo():
    global i_am_global
    i_am_global = 10
    print(i_am_global)

foo()
print(i_am_global)

# outputs
```

```python
def foo():
    i_am_non_local = 5

    def bleep():
        i_am_non_local = 10

    def oof():
        nonlocal i_am_non_local
        i_am_non_local = 20

    def bop():
        global i_am_non_local
        i_am_non_local = 30

    bleep()
    print('After bleep:', i_am_non_local)
    oof()
    print('After oof:', i_am_non_local)
    bop()
    print('After bop:', i_am_non_local

foo()
print('Globally:', i_am_non_local)



```

writable_nonlocals.py hosted with ❤ by GitHub view raw

Writable Non-locals

The program gives the following output:

```
After bleep: 5
After oof: 20
After bop: 20
Globally: 30
```

writable_nonlocals_res.py hosted with ❤ by GitHub view raw

Writable Non-locals Result

A few things to take note about this result:

- `bleep()` does not alter the variable since it creates its own copy in its namespace
- `oof()` binds itself to the non-local variable
- `bop()` creates a *global* variable. This is why we have access to the variable outside the function. *They have the same names but reside in different namespaces.*

By default, python protects you from modifying outer variables from inner variables so that your program stays predictable. Use the `global` and `nonlocal` keywords only if you have specific use-cases for it. Most of the time, you won't!

Namespaces and scopes are tricky, but mastering them is essential for any Python programmer.

## Further Reading

- The Python Class Documentation offers an extensive (and more technical) explanation of scopes and namespaces.
- Functional Programming in Python provides reasons why you wouldn't want to use the *global* and *nonlocal* keywords. If you are coming from C or Java, this is a good read.

## More of My Work

If you like what you read, you might be interested in my other articles:

- Writing Code People Want to Read
- A Beginner's Guide to Python and its Quirks
- Pandas Hacks: read_clipboard()

## About Adrian Perea

*Part-time Data Scientist, part-time Tech Writer, full-time Nerd. I find joy in teaching Machine Learning and Programming in manageable,* ***byte****-sized pieces. Yes, I'm also* *hilarious. Coffee addict. Sucker for data visualization. Willing to give my 1's and 0's for the next best Tree Map.*

*Follow me on Twitter and LinkedIn.*

Python Programming Machine Learning Data Science

166 | 3