Scope and user-defined functions

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Hugo Bowne-Anderson
Instructor

datacamp

Crash course on scope in functions

- Not all objects are accessible everywhere in a script
- Scope - part of the program where an object or name may be accessible
  - Global scope - defined in the main body of a script
  - Local scope - defined inside a function
  - Built-in scope - names in the pre-defined built-ins module

Global vs. local scope (1)

```
def square(value):
    """Returns the square of a number."""
    new_val = value ** 2
    return new_val
square(3)
```

3. Global vs. local scope (1)
If we then try to access the variable name new_val after function execution, the name is not accessible. This is because it was defined only within the local scope of the function. The name new_val was not defined globally.

```
9
```

```
new_val
```

```
<hr />---------------------------------------------------------------
NameError                          Traceback (most recent call last)
<ipython-input-3-3cc6c6de5c5c> in <module>()
<hr />-> 1 new_value
NameError: name 'new_val' is not defined
```

Global vs. local scope (2)

```
new_val = 10

def square(value):
    """Returns the square of a number."""
    new_val = value ** 2
    return new_val
square(3)
```

```
9
```

```
new_val
```

```
10
```

4. Global vs. local scope (2)
Now what if we define the name globally before defining and calling the function? In short, any time we call the name in the global scope, it will access the name in the global, such as you see here. Any time we call the name in the local scope of the function, it will look first in the local scope. That's why calling square(3) results in 9 and not 10. If Python cannot find the name in the local scope, it will then and only then look in the global scope.

Global vs. local scope (3)

```
new_val = 10

def square(value):
    """Returns the square of a number."""
    new_value2 = new_val ** 2
    return new_value2
square(3)
```

```
100
```

```
new_val = 20

square(3)
```

```
400
```

5. Global vs. local scope (3)
Here, for example, we access new_val defined globally within the function square. Note that the global value accessed is the value at the time the function is called, not the value when the function is defined. Thus, if we re-assign new_val and call the function square, we see that the new value of new_val is accessed. To recap, when we reference a name, first the local scope is searched, then the global. If the name is in neither, then the built-in scope is searched.

Global vs. local scope (4)

```
new_val = 10

def square(value):
    """Returns the square of a number."""
    global new_val
    new_val = new_val ** 2
    return new_val
square(3)
```

6. Global vs. local scope (4)
Now what if we want to alter the value of a global name within a function call? This is where the keyword global comes in handy. To look at how it works, let's look at another example. Within the function definition, we use the keyword global followed by the name of the global variable that we wish to access and alter. For example, here we change new_val to its square. The function call works as one would expect. Now calling new_val, we see that the global value has indeed been squared by running the function square.

```
100
```

```
new_val
```

```
100
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Nested functions

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Hugo Bowne-Anderson
Instructor

datacamp

Nested functions (1)

```
def outer( ... ):
    """ ... """
    x = ...

    def inner( ... ):
        """ ... """
        y = x ** 2
    return ...
```

2. Nested functions (1)
What if we have a function inner defined within another function outer and we reference a name x in the inner function? The answer is intuitive: Python searches the local scope of the function inner, then if it doesn't find x, it searches the scope of the function outer, which is called an enclosing function because it encloses the function inner. If Python can't find x in the scope of the enclosing function, it only then searches the global scope and then the built-in scope. But whoa, hold on there for a second, why are we even nesting functions?

Nested functions (2)

```
def mod2plus5(x1, x2, x3):
    """Returns the remainder plus 5 of three values."""

    new_x1 = x1 % 2 + 5
    new_x2 = x2 % 2 + 5
    new_x3 = x3 % 2 + 5

    return (new_x1, new_x2, new_x3)
```

3. Nested functions (2)
There are a number of good reasons to do so. Let's say that we want to use a process a number of times within a function. For example, we want a function that takes 3 numbers as parameters and performs the same function on each of them. One way would be to write out the computation 3 times

Nested functions (3)

```
def mod2plus5(x1, x2, x3):
    """Returns the remainder plus 5 of three values."""

    def inner(x):
        """Returns the remainder plus 5 of a value."""
        return x % 2 + 5

    return (inner(x1), inner(x2), inner(x3))
```

```
print(mod2plus5(1, 2, 3))
```

```
(6, 5, 6)
```

Returning functions

```
def raise_val(n):
    """Return the inner function."""

    def inner(x):
    """Raise x to the power of n."""
        raised = x ** n
        return raised

    return inner
```

```
square = raise_val(2)
cube = raise_val(3)
print(square(2), cube(4))
```

```
4 64
```

5. Returning functions
Now look at what raise_vals returns: it returns the inner function inner! raise_vals takes an argument n and creates a function inner that returns the nth power of any number. That's a bit complicated and will be clearer when we use the function raise_vals. Passing the number 2 to raise_vals creates a function that squares any number. Similarly, passing the number 3 to raise_vals creates a function that cubes any number. One interesting detail: when we call the function square, it remembers the value n=2, although the enclosing scope defined by raise_val and to which n=2 is local, has finished execution. This is a subtlety referred to as a closure in Computer Science circles and shouldn't concern you too much. It is worth mentioning, however, as you may encounter it out there in the wild.

Using nonlocal

```
def outer():
    """Prints the value of n."""
    n = 1

    def inner():
        nonlocal n
        n = 2
        print(n)

    inner()
    print(n)
```

6. Using nonlocal
Recall from our discussion of scope that you can use the keyword global in function definitions to create and change global names; ***similarly, in a nested function, you can use the keyword nonlocal to create and changes names in an enclosing scope.***
In this example, we alter the value of n in the inner function; because we used the keyword nonlocal, it also alter the value of n in the enclosing scope. This is why calling the function outer prints the value of n as determined within the function inner.

```
outer()
```

```
2
2
```

Scopes searched

- Local scope
- Enclosing functions
- Global
- Built-in

7. Scopes searched
To summarize: name references search at most four scopes, the local scope, then those of enclosing functions, if there are any; then global, then built-in. This is known as the LEGB rule, where L is for local, E for enclosing, G for global and B for built-ins! Also, remember that assigning names will only create or change local names, unless they are declared in global or nonlocal statements using the keyword global or the keyword nonlocal, respectively.

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Default and flexible arguments

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Hugo Bowne-Anderson
Instructor

datacamp

You'll learn:

- Writing functions with default arguments
- Using flexible arguments
  - Pass any number of arguments to a functions

Add a default argument

```
def power(number, pow=1):
    """Raise number to the power of pow."""
    new_value = number ** pow
    return new_value

power(9, 2)
```

```
81
```

```
power(9, 1)
```

```
9
```

```
power(9)
```

```
9
```

Flexible arguments: *args (1)

```
def add_all(*args):
    """Sum all values in *args together."""

    # Initialize sum
    sum_all = 0

    # Accumulate the sum
    for num in args:
        sum_all += num

    return sum_all
```

4. Flexible arguments: *args (1)
Lets now look at flexible arguments: let's say that you want to write a function but aren't sure how many arguments a user will want to pass it; for example, a function that takes floats or ints and adds them all up, irrespective of how many there are. Enter flexible arguments! In this example, we write the function that sums up all the arguments passed to it. In the function definition, we use the parameter star followed by args: this then turns all the arguments passed to a function call into a tuple called args in the function body; then, in the function body, to write our desired function, we initialize our sum sum_all to 0, loop over the tuple args and add each element of it successively to sum_all and then return it.

Flexible arguments: *args (2)

```
add_all(1)
```

```
1
```

```
add_all(1, 2)
```

```
3
```

```
add_all(5, 10, 15, 20)
```

```
50
```

Flexible arguments: **kwargs

```
print_all(name="Hugo Bowne-Anderson", employer="DataCamp")
```

```
name: Hugo Bowne-Anderson
employer: DataCamp
```

6. Flexible arguments: **kwargs
You can also use a double star to pass an arbitrary number of keyword arguments, also called kwargs, that is, arguments preceded by identifiers. We'll write such a function called print_all that prints out the identifiers and the parameters passed to them as you see here.

Flexible arguments: **kwargs

```
def print_all(**kwargs):
    """Print out key-value pairs in **kwargs."""

    # Print out the key-value pairs
    for key, value in kwargs.items():
        print(key + \": \" + value)
```

```
print_all(name="dumbledore", job="headmaster")
```

```
job: headmaster
name: dumbledore
```

7. Flexible arguments: **kwargs
Now to write such a function, we use the parameter kwargs preceded by a double star. This turns the identifier-keyword pairs into a dictionary within the function body. Then, in the function body all we need to do is to print all the key-value pairs stored in the dictionary kwargs. Note that it is NOT the names args and kwargs that are important when using flexible arguments, but rather that they're preceded by a single and double star, respectively.

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Bringing it all together

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Hugo Bowne-Anderson
Instructor

datacamp

Next exercises:

- Generalized functions:
  - Count occurrences for any column
  - Count occurrences for an arbitrary number of columns

Add a default argument

```
def power(number, pow=1):
    """Raise number to the power of pow."""
    new_value = number ** pow
    return new_value
```

```
power(9, 2)
```

```
81
```

```
power(9)
```

```
9
```

Flexible arguments: *args (1)

```
def add_all(*args):
    """Sum all values in *args together."""

    # Initialize sum
    sum_all = 0

    # Accumulate the sum
    for num in args:
        sum_all = sum_all + num

    return sum_all
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)