Scope and user-defined functions

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Hugo Bowne-Anderson
Instructor

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

Once the execution of a function is done, any name inside the local scope ceases to exist, which means you cannot access those names anymore outside of the function definition.

such as print and sum
import builtins
dir(builtins): print a list of all the names in the module builtins

Global vs. local scope (1)

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

```
9
```

```
new_val
```

The name is not accessible.
This is because it was defined only within the local scope of the function.

```
<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)
```

Any time we call the name in the local scope of the function, it will look first in the local scope.
If Python cannot find the name in the local scope, it will then and only then look in the global scope.

```
9
```

```
new_val
```

Any time you call the name in the global scope, it will access the name in the global.

```
10
```

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)
```

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.

```
100
```

```
new_val = 20

square(3)
```

Thus, if we re-assign new_val and call the function square, we see the new value of new_val is accessed.
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.

```
400
```

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)
```

You can use the keyword global within a function to alter the value of a variable defined in the global scope.

```
100
```

```
new_val
```

The global value has been squared by running the function square.

```
100
```

Let's practice!

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Nested functions

PYTHON DATA SCIENCE TOOLBOX (PART 1)

Hugo Bowne-Anderson
Instructor

Nested functions (1)

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

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

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.

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)
```

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
```

A closure is a nested function which has access to a free variable from an enclosing function that has finished its execution. Three characteristics of a Python closure are:
1. it is a nested function
2. it has access to a free variable in outer scope
3. it is returned from the enclosing function

return the nth power of any number

Functions can be assigned to variables, stored in collections, created and deleted dynamically, or passed as arguments.

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

At this moment, the raise_val function has finished its execution and n is a local variable. However, the inner() closure still has access to the n variable.

```
4 64
```

Using nonlocal

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

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

    inner()
    print(n)
```

It also alters the value of n in the enclosing scope.

A free variable is a variable that is not bound in the local scope. In order for closures to work with immutable variables such as numbers and strings, we have to use the nonlocal keyword.
The nonlocal keyword allows us to modify a variable with immutable type in the outer function scope.

```
outer()
```

```
2
2
```

Scopes searched

LEGB rule

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

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

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)

You want to write a function but aren't sure how many arguments a user will want to pass it.

```
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
```

Flexible arguments: *args (2)

```
add_all(1) a tuple
```

```
1
```

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

```
3
```

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

```
50
```

Flexible arguments: **kwargs

pass an arbitrary number of keyword arguments

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

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

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)
```

This turns the identifier-keyword pairs into a dictionary within the function body.

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

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

Note that it is NOT the same args and kwargs that are important when using flexible arguments, but rather that they are 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

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)