Namespaces, Scope, & Modules/Packages

D. A. Sirianni

CHEM 4803/8843 DR

School of Chemistry & Biochemistry
Georgia Institute of Technology

27 August 2019

Overview

Primer

What will the following code, `scope_test.py`, print?

```python
some_name = 'Bob'

def print_name():
    some_name = 'Alice'
    print(f'In here, the name is {some_name}')

print(f'Out here, the name is {some_name}')
print_name()
```

Primer

What will the following code, scope_test.py, print?

```python
some_name = 'Bob'

def print_name():
    some_name = 'Alice'
    print(f'In here, the name is {some_name}')

print(f'Out here, the name is {some_name}')
print_name()
```

Answer:

```
$ python scope_test.py

Out here, the name is Bob.
In here, the name is Alice.
```

Some Definitions

Namespace

A *namespace* is a mapping of name to object, as in a Python dictionary:

```python
space1 = {'name1': var1, 'name2': var2, ...}
space2 = {'name0': var1, 'name3': var2, 'name4': var4, ...}
```

Note: Namespaces are completely independent, and their names have no relation to one another. Furthermore, they can have different lifetimes.

Scope

Scope is the textual region of a Python program where a namespace is directly accessible.

Scope Regions: Local & Global

```python
# Here is the global (G) scope
global_var = "This is a global variable"

def my_function():
    # This is the local (L) scope
    local_var = "This is a local variable"
```

Scope Regions: Local & Global

```python
# Here is the global (G) scope
global_var = "This is a global variable"

def my_function():
    # This is the local (L) scope
    local_var = "This is a local variable"
```

Namespace Precedence

When Python searches for the object associated with a particular name, it proceeds from $L \to G$; this is referred to as their *namespace precedence*.

Scope Regions: Enclosing & Built-In

```python
# Here is the global (G) scope
global_var = "This is a global variable"

def outer_function():
    # This is the enclosing (E) space
    enclosing_var = "This is a nonlocal variable"

    def inner_function():
        # This is the local (L) space
        local_var = "This is a local variable"

# Some names are built-in (B) to Python
print(type(global_var))

def type():
    print("typetypetypetype")

type()
```

Namespace Precedence: The LEGB Rule

```python
def set():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

set()
print("In global scope:", spam)
```

Namespace Precedence: The LEGB Rule

```
$ python legb.py

After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Namespace Precedence: The LEGB Rule

```
$ python legb.py

After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

The LEGB Rule

Object namespaces have the following order of precedence:

$$\text{Local} \rightarrow \text{Enclosed} \rightarrow \text{Global} \rightarrow \text{Built-In}$$

this is referred to as the *LEGB Rule*.

Organizing Code

Let's pretend we work for Google Maps. (cool, right?)

We have written the following code to compute the distance between two points, (x_1, y_1) and (x_2, y_2) in Cartesian space:

```python
def distance(coords1, coords2):
    total = 0.0
    for k in list(len(coords1)):
        total += (coords1[k] - coords2[k]) ** 2
    return (total ** 0.5)
```

Our colleague wants to write a different function called `distance()` which takes user input for selecting the points on a map and returns the distance between them.

Organizing Code

What are some possible ways she could do this?
(Hint: Just think about code components, not how to actually collect the user input, etc.)

- Copy our function within her function
- Copy code from our function directly into hers, don't bother defining new function

Function-ception!

```python
def distance(user_click_1, user_click_2):
    # Our distance function, copied and pasted here
    def distance(coords1, coords2):
        total = 0.0
        for k in list(len(coords1)):
            total += (coords1[k] - coords2[k]) ** 2
        return (total ** 0.5)

    # Colleague's code to collect user clicks
    # and convert latitude-longitude to (x, y)
    ...
    user_cart1 = [user_x_1, user_y_1]
    user_cart2 = [user_x_2, user_y_2]
    cartesian_distance = distance(user_cart1, user_cart2)

    # Colleague's code to convert from (x, y) distance to map distance
    ...
    return map_distance
```

Modules: A Better Solution

Instead of literally copying-and-pasting each others' code into the same file, Python allows us to better organize our code by splitting it up into different files, called *modules*. A collection of modules is called a *package*.

If we save our distance function inside a file called `module_test.py`, our colleague can use our code by *importing* our module:

```python
# Most simple
import module_test

# Give our module a nickname
import module_test as mod

# Import a particular function from our module
from module_test import distance

# Import everything from module_test into current file; just like copy-paste
from module_test import *
```

Using Modules

Modular Solution

```python
def distance(user_click_1, user_click_2):
    # Our distance function, imported and used here
    from module_test import distance

    # Colleague's code to collect user clicks
    # and convert latitude-longitude to (x, y)
    ...
    user_cart1 = [user_x_1, user_y_1]
    user_cart2 = [user_x_2, user_y_2]
    cartesian_distance = distance(user_cart1, user_cart2)

    # Colleague's code to convert from (x, y) distance to map distance
    ...
    return map_distance
```

After a few months of development...

```python
def distance(user_click_1, user_click_2):
    # Our distance function, imported and used here
    from module_test import distance
    ...
    cartesian_distance = distance(user_cart1, user_cart2)
    ...
    return map_distance

def similarity_score(feature_1, feature_2):
    # Convert features to Cartesian coordinates
    ...
    # Compute the distance using our function
    from module_test import distance

    cartesian_distance = distance(feature_cart_1, feature_cart_2)
    ...
```

Colleague Refactors Her Code...

Wouldn't it be easier to only import *module_test* once?

```
# Our distance function, imported once and used often
from module_test import distance

def distance(user_click_1, user_click_2):
    ...
    cartesian_distance = distance(user_cart1, user_cart2)
    ...

def similarity_score(feature_1, feature_2):
    ...
    cartesian_distance = distance(feature_cart_1, feature_cart_2)
    ...
```

What problems have we introduced here?

Module Namespaces

Each module has its own namespace when it is imported, unless otherwise specified:

```python
import module               # Imports module as separate namespace
import module as mod        # Gives module namespace a nickname
from module import func     # Imports function `func` into current namespace
from module import *        # Imports everything into current namespace
```

Module namespaces can be accessed using "dot syntax:"

```python
import module as mod

# Call function `func` from inside `module`'s namespace
result = mod.func()
```

Refactored Code Preserving Namespaces

```python
# Our module, imported as its own namespace
import module_test as mod

def distance(user_click_1, user_click_2):
    ...
    cartesian_distance = mod.distance(user_cart1, user_cart2)
    ...

def similarity_score(feature_1, feature_2):
    ...
    cartesian_distance = mod.distance(feature_cart_1, feature_cart_2)
    ...
```

Scientific Computing...

...is a rapidly growing, multidisciplinary field that uses advanced computing capabilities to understand and solve complex problems. It is an area of science which spans many disciplines, but at its core it involves the development of models and simulations to understand natural systems.[1]

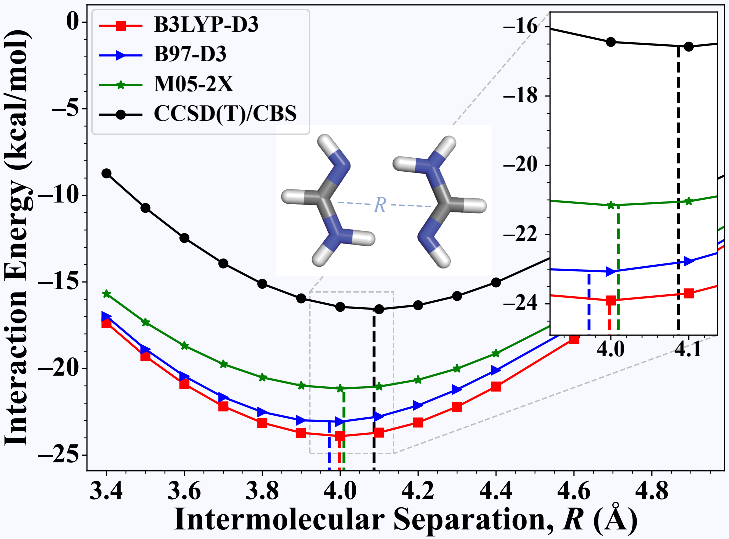
- Using industry-specific, fully-featured programs
- Integrating existing/available tools to solve specific problems
- Writing custom software routines to perform simulations, etc.

[1] From Wikipedia, article "Computational Science." `https://en.wikipedia.org/wiki/Computational_science`

Python Libraries for Scientific Computing

- SciPy: Diverse capabilities for scientific computing
- NumPy: Numerical linear algebra routines
- Pandas: Database capabilities
- Scikits: Domain-specific tools in a variety of fields
- Matplotlib: Generates publication-quality plots & figures
- Keras & Tensorflow: Neural Networks made easy!
- Many others: `https://wiki.python.org/moin/NumericAndScientific`

An Illustrative Case Study: Curve Minima Interpolation


B3LYP-D3
B97-D3
M05-2X
CCSD(T)/CBS
Interaction Energy (kcal/mol)
Intermolecular Separation, R (Å)
R