Python Programming for Data Science

Week 39, Monday

- Namespaces
- Modules

Namespaces and scopes

Namespaces and Scopes

Variables defined inside a function do not conflict with variables defined outside

```
a = 2
def f():
    a=3
    return a

print(f())

print(a)
```

output

```
3

2
```

Namespaces and Scopes

Variables defined inside a function do not conflict with variables defined outside

```
a = 2
def f():
    a=3
    return a

print(f())

print(a)
```

output

```
3

2
```

Q: How does Python know which value a variable name refers to?

Namespaces and Scopes

Variables defined inside a function do not conflict with variables defined outside

```
a = 2
def f():
    a=3
    return a

print(f())

print(a)
```

output

```
3

2
```

Q: How does Python know which value a variable name refers to?

A: through namespaces and scope rules.

Namespaces

Every function call results in the creation of a new *namespace*

Namespaces

Every function call results in the creation of a new *namespace*

Namespace: Mapping from names to values

Namespaces

Every function call results in the creation of a new *namespace*

Namespace: Mapping from names to values

When the function call ends, the namespace disappears.

Namespaces

Every function call results in the creation of a new *namespace*

Namespace: Mapping from names to values

When the function call ends, the namespace disappears.

When you define a new variable, it is done in the current namespace. It might shadow over a variable in the global namespace, but it will not overwrite it.

Scope

The *scope* of a variable is a term used to describe where in the program a particular variable can be used.

When you refer to a variable, Python uses scope rules to determine which variable you mean:

Scope

The *scope* of a variable is a term used to describe where in the program a particular variable can be used.

When you refer to a variable, Python uses scope rules to determine which variable you mean:

1. Look in current name space

Scope

The *scope* of a variable is a term used to describe where in the program a particular variable can be used.

When you refer to a variable, Python uses scope rules to determine which variable you mean:

1. Look in current name space
2. Look in enclosing namespaces (nested functions – ignore this)

Scope

The *scope* of a variable is a term used to describe where in the program a particular variable can be used.

When you refer to a variable, Python uses scope rules to determine which variable you mean:

1. Look in current name space
2. Look in enclosing namespaces (nested functions – ignore this)
3. Look in global namespace (the namespace of the file)

Scope

The *scope* of a variable is a term used to describe where in the program a particular variable can be used.

When you refer to a variable, Python uses scope rules to determine which variable you mean:

1. Look in current name space
2. Look in enclosing namespaces (nested functions – ignore this)
3. Look in global namespace (the namespace of the file)
4. Look in built-in namespace

Scope - example

Same example as before:

```
a = 2          # Insert "a" into namespace
def f():       # When f is called - a new namespace is created
    a=3        # Insert a into temporary namespace of function
    print(a) # Looking for "a" in current namespace...found a=3

print(f())     # "a" inside function refers to 3

print(a)       # "a" outside function is still 2.
```

output

```
3

2
```

Namespace & Scope - Exercise

Consider the following case

```
x = 1

def f(x):
    z = x

for i in range(4):
    x = i
    y = x
    f(x)

print(x)
print(y)
print(z)
```

Without running the code - guess what the printed values of x,y,z are. Why?

Modules

Modules

Another level of structural building block

Like functions are a collection of statements, modules are a collection of functions, statements and classes

Modules

Another level of structural building block

Like functions are a collection of statements, modules are a collection of functions, statements and classes

Every `.py` file is a module. The name of the module is the name of the file without the `.py` extention.

Modules - importing

Modules define a namespace

Modules - importing

Modules define a namespace

By importing a module, you gain access to this name space

Modules - importing

Modules define a namespace

By importing a module, you gain access to this name space

Importing can be done in two ways: `import` or `from`

import

```
import module_name
```

imports the module as a single object in your namespace

```
import random
random.randint(1,6)     # all names are accessed through "random"
4
```

import

```
import module_name
```

imports the module as a single object in your namespace

```
import random
random.randint(1,6)     # all names are accessed through "random"
4
```

from ... import

```
from module_name import names
```

import individual names from the module into your current namespace

```
from random import randint
randint(1,6)    # the randint name has been imported into the namespace
```

`from ... import *`

```
from module_name import *
```

This will import all available names into your current namespace.

```
from random import *
randint(1,6)
```

`import` vs `from`

Which should you use?

`import` vs `from`

Which should you use?

- `from` makes the important objects more easily available

`import` vs `from`

Which should you use?

- `from` makes the important objects more easily available
- `import` keeps the namespaces neatly separated

`import` vs `from`

Which should you use?

- `from` makes the important objects more easily available
- `import` keeps the namespaces neatly separated

By using `from`, you risk name-clashes. Especially when using `from...import *`.

`import` vs `from`

Which should you use?

- `from` makes the important objects more easily available
- `import` keeps the namespaces neatly separated

By using `from`, you risk name-clashes. Especially when using `from...import *`.

Therefore, `from...import *` is OK for testing purposes, but for real programs it is better to specify which names you want to import using `from`, or keep them in their own object using `import`.

as

Both the `import` and `from` import techniques support the `as` keyword

This allows you to import a module under a different name

```
import random as rnd
rnd.randint(1,6)
```

Importing modules - Exercise

- Define a function called `randint` that simply prints "hello"
- Import the randint function from the random module in two different ways that don't overwrite your existing function. Check whether your original `randint` function still works.
- Import everything from the random module using `from ... import *` and check whether your original `randint` function still works.

Importing - Where does python look for modules?

Whenever you write an import statement, Python will look through a list of directories to find it

Importing - Where does python look for modules?

Whenever you write an import statement, Python will look through a list of directories to find it

You can inspect (and change) this list through the `sys` module

python_path_test.py

```
...
import sys          # Importing sys module
print(sys.path)
```

output

```
['/home/lpp', '/home/lpp', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-i3
```

Importing - Where does python look for modules?

Whenever you write an import statement, Python will look through a list of directories to find it

You can inspect (and change) this list through the `sys` module

python_path_test.py

```
...
import sys          # Importing sys module
print(sys.path)
```

output

```
['/home/lpp', '/home/lpp', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-i3
```

Alternatively, you can set it from Bash by setting the `PYTHONPATH` environment variable (e.g. in your .bashrc)

Packages

Larger libraries are sometimes organized as a "package", which is basically just directory of module files.

Packages

Larger libraries are sometimes organized as a "package", which is basically just directory of module files.

Importing a package:

```
import Bio.PDB.PDBParser
```

This means that somewhere on the python path, there is a directory called `Bio`, under which there is a directory called `PDB`, which containes various Python modules, one of which is called `PDBParser.py`.

Packages

Larger libraries are sometimes organized as a "package", which is basically just directory of module files.

Importing a package:

```
import Bio.PDB.PDBParser
```

This means that somewhere on the python path, there is a directory called `Bio`, under which there is a directory called `PDB`, which containes various Python modules, one of which is called `PDBParser.py`. Note how `.` is used as directory separator.

Packages

Larger libraries are sometimes organized as a "package", which is basically just directory of module files.

Importing a package:

```
import Bio.PDB.PDBParser
```

This means that somewhere on the python path, there is a directory called `Bio`, under which there is a directory called `PDB`, which containes various Python modules, one of which is called `PDBParser.py`. Note how `.` is used as directory separator.

For this course, you will not be required to write python packages yourself. But you should be able to import them.

Modules - Exercise

1. Create a file called `coin_toss_module.py`, containing our old `coin_toss` function:

```
import random

def coin_toss(heads_prob=0.5):
    '''Return "heads" or "tails" with a specified  probability'''
    x = random.random()
    if x < heads_prob:
        return "heads"
    else:
        return "tails"
```

2. Create a new Python file called `coin_toss_test.py`
3. Call the `coin_toss` function from within the `coin_toss_test.py` file.
4. Create a directory called `my_modules` in your current directory, and move the `coin_toss_module.py` file to this new directory. Verify that your import in `coin_toss_test.py` no longer works. Now fix it by adding the `my_modules` directory to the `sys.path`.