Python Programming for Data Science

Week 38, Monday

Looping revisited

Namespaces

Modules

A popquiz

- What is an *argument*?
- How do you define a function?
- How do you call a function?
- What is a return value?
- What is a breakpoint, and how do you set one?
- Does the person calling the function need to provide values for all arguments?
- Is the order of arguments important?
- What is a doc-string?

Looping revisited

Recall: while loops and for loops

While loop

```
i = 0
while i < 4:
    print(i)
    i += 1
```

For loop:

```
for i in [0,1,2,3]
    print(i)
```

For-loops are generally more convenient: We don't have to create a counting variable, and we don't run the risk of creating an infinite loops.
But how do we use for loops to iterate over a range of numbers? Writing "[0,1,2,3]" is not a scalable solution.

The range function

The range function allows you to create a list of numbers running from start to stop-1 with some step size.

```
range([start], stop, [step-size])
```

You can then iterate through this list using the for statement:

```
for i in range(3):        # range(3) ->
[0,1,2]
    print(i+1)
```

```
output
1
2
3
```

Note that start and step-size are optional parameters.

The for loop - with indices

If we need to know the index, we can iterate over indices instead:

Original

```
container = ["a", "b", "c"]
for val in container:
    # I don't know the
index
    print("Value at index ?
is: ", val)
```

Iterating over indices

```
container = ["a", "b", "c"]
for i in
range(len(container)):
    val = container[i]
    # Now I know the
current index i
    # I can get the value
by looking it up
    print("Value at index",
i, "is:", val)
```

There is a built-in function `enumerate`, that make this easier:

```
container = ["a", "b", "c"]
for i,val in enumerate(container):          # Note the
i,val tuple after for
    # Now I know the current index i AND I have access to
the value directly
    print("Value at index", i, "is:", val)
```

Loops - else statement

Loops can have an else clause, which will execute if the loop finishes normally (without a break statement).

```
>>> l = range(1, 10)
>>> for i in l:
...     if i == 12:
...         break
... else:
...     print(" Element not found ")
...
Element not found
```

Nested loops

Nested data structures often require nested loops:

```
c = [[1,2,3], [4,5,6], [7,8,9]]
for inner_list in c:                    # Iterate over outer
list
    print("Starting on new list:", inner_list)
    for entry in inner_list:            # Iterate over inner
list
        print(entry)
```

```
output
Starting on new list: [1, 2, 3]
1
2
3
Starting on new list: [4, 5, 6]
4
5
6
Starting on new list: [7, 8, 9]
7
8
9
```

Loops - exercise

1. Create the following list: [6,9,4,8,7,1,2].
2. Write a while loop that prints each element in the list.
3. Write a for loop that prints each element in the list.
4. Write a loop that prints all elements that are larger than their left neighbour (for the above example: 9, 8, 2). Which type of loop is best suited for this purpose?

Loops - exercise - solution

1. Create the following list: [6,9,4,8,7,1,2].

```
number_list = [6,9,4,8,7,1,2]
```

2. Write a while loop that prints each element in the list.

```
i = 0
while i < len(number_list):
    print(number_list[i])
    i += 1
```

3. Write a for loop that prints each element in the list.

```
for entry in number_list:
    print(entry)
```

Loops - exercise - solution (2)

4. Write a loop that prints all elements that are larger than their left neighbour (for the above example: 9, 8, 2). Which type of loop is best suited for this purpose?

```
for i in range(1, len(number_list)):          # i starts
at one
    if number_list[i] > number_list[i-1]:    # compare
to left-neighbor
        print(number_list[i])
```

   Alternatively, there is a way of automatically getting both value and index:

```
for i,entry in enumerate(number_list): # enumerate
returns (i,value) pairs
    if i > 0 and entry > number_list[i-1]:
        print(entry)
```

Moral of the story: even when you need indices, its still easier to use `for` than `while`.

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
```

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*

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:

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?

Namespace & Scope - Exercise - solution

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

A. x and y are both 3, while z is undefined (only defined within the namespace of the function).

Take home message: in Python, loops do not define their own namespace.

Modules

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
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
```

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?

- `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 modules - Exercise - solution

- Define a function called `randint` that simply prints "hello"

module_import_test.py

```
def randint():
    print("hello")
```

Importing modules - Exercise - solution (2)

- 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.

```
...
module_import_test1.py

# 1. Importing through random module object
import random
print(random.randint(1,6))

# 2. Using "as" to avoid name conflict
from random import randint as rnd_randint
print(rnd_randint(1,6))

randint()   # Does our own function still work?
```

```
output
4
5
hello
```

Importing modules - Exercise - solution (3)

- Import everything from the random module using `from ... import *` and check whether your original `randint` function still works.

```
...
module_import_test2.py

# Importing everything from random into our namespace
from random import *

# Attempting to call our own randint function
randint()
```

```
output
Traceback (most recent call last):
  File "/home/lpp/module_import_test2.py", line 7, in
    randint()
TypeError: randint() takes exactly 3 arguments (1
given)
```

Our function has been overwritten.

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-i386-linux-gnu',
'/usr/lib/python2.7/lib-tk', '/usr/lib/python2.7/lib-old',
'/usr/lib/python2.7/lib-dynload',
'/usr/local/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages',
'/usr/lib/python2.7/dist-packages/PILcompat',
'/usr/lib/python2.7/dist-packages/gtk-2.0',
'/usr/lib/pymodules/python2.7', '/usr/lib/python2.7/dist-
packages/ubuntu-sso-client']
```

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`.

Modules - Exercise - solution

coin_toss_module.py

```
import random

def coin_toss(heads_prob=0.5):  # define function with optional argument
    x = random.random()
    if x < heads_prob:          # determine whether it is heads or tails
        return "heads"
    else:
        return "tails"
```

coin_toss_test.py

```
import coin_toss_module
result = coin_toss_module.coin_toss()
print(result)
```

output

```
'heads'
```

Modules - Exercise - solution (2)

Move the module to the `my_modules` directory:

```
my_modules/coin_toss_module.py
import random

def coin_toss(heads_prob=0.5):  # define function with optional argument
    x = random.random()
    if x < heads_prob:          # determine whether it is heads or tails
        return "heads"
    else:
        return "tails"
```

Now: this doesn't work:

```
coin_toss_test.py
import coin_toss_module
result = coin_toss_module.coin_toss()
print(result)
```

```
output
Traceback (most recent call last):
  File "/Users/wb/teaching/ppds2020/code/playground/coin_toss_test.py", line 3, in <module>
    import coin_toss_module
ModuleNotFoundError: No module named 'coin_toss_module'
```

Modules - Exercise - solution (2)

Fixing it by adding my_modules to the path

coin_toss_test.py

```
import sys
sys.path.append('my_modules')
import coin_toss_module      # This now works
result = coin_toss_module.coin_toss()
print(result)
```

output

```
heads
```

Speaker notes