Python

How You Can Do More Interesting Things With Python (Part II)

Python `for` Statement

`for` *<target-list>* `in` *<iterator>*`:`
    statement-block

```
my_dict = { ‘k1’: 1, ‘k2’: 2 }
for (k,v) in my_dict.items():
    print( “Key = %s, value = %s’ % ( k, v ) )
```

- For the sake of completeness, this is one way to iterate over a dict.
- Iteration of a dict directly returns the keys.
- It possible to get the values directly

Python `for` Statement

`for` *<target-list>* `in` *<iterator>*`:`
    statement-block

Do not forget that:

- `continue` returns to the top of the loop and executes the *<iterator>*.
- `break` ends the loop and starts execution after the *statement-block*.

Python `while` Statement

```
while <boolean>:
   statement-block
```

- This is much simpler. Runs until the *boolean* is `False`.
- I am going to skip examples here.

Python `else` on Iteration Statements

```
while <boolean>:
   statement-block
else:
   statement-block
```

The `else` executes if the loop terminates **without** using `break.`

```
>>> while False:
...     break
... else:
...     print 'else'
...
else
```

Python `else` on Iteration Statements

```
for i in xrange( 0, 10 ):
    if i == 5:
        break
else:
    print ‘else’
```

Since the `else` executes if the loop terminates **without** using `break`, we can see pretty clearly that the `else` **will not** execute in this example.

Python `else` on Iteration Statements

```
for v in my_list:
    if v == looking_for:
        <code for finding>
        break
else:
    <code when not found>
```

vs

```
found = False
for v in my_list:
    if v == looking_for:
        found = True
if found:
    <code for finding>
else:
    <code when not found>
```

The `else` portion of the statement can often eliminate the need for flag variables in the loop that describe how the loop exited.

Python `else` on Iteration Statements

```
for v in my_list:
    if v == looking_for:
        <code for finding>
else:
    <code when not found>
```

- Watch the indent level to see what statement the else is associated with.
- In this case, the `else` is clearly associated with the for.
- Keep the code blocks pretty short so the indent can be easily read.
- Use functions to keep the code short.

Python For Statement

From the last session, question on processing multiple lists:

```
#!/usr/bin/python

from __future__ import print_function

l1 = [ 1,2,3,4,5 ]
l2 = [ 11,12,13,14,15]
l3 = [ 21,22,23,24,25]

for l in ( l1, l2, l3 ):
    for i in l:
        print( "%d" % i, end=',' )
print( '\n' )
```

Will print:

```
1,2,3,4,5,11,12,13,14,15,21,22,23,24,25,
```

Python For Statement

From the last session, question on processing multiple lists:

```
#!/usr/bin/python
from __future__ import print_function

v = []
v.append( [ 1,2,3,4,5 ] )
v.append( [ 11,12,13,14,15] )
v.append( [ 21,22,23,24,25] )

for l in v:
    for i in l:
        print( "%d" % i, end=',' )
print( '\n' )
```

Will print:

```
1,2,3,4,5,11,12,13,14,15,21,22,23,24,25,
```

Python String Formatting

The old, somewhat easier way:

“Format string with `%` directives” `%` (`<a-tuple-with-values-in-order>`)

or

“Format string with one `%` directive” `%` `<one-value>`

Python String Formatting

```
>>> "A test %d %s" % ( 1, "Yep" )
>>> 'A test 1 Yep'
```

```
>>> "A test %d" % 1
>>> 'A test 1'
```

```
>>> "A test %d %s" % 1, "yep"
>>> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: not enough arguments for format
string
```

Python String Formatting

New Style Formatting:

```
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
```

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N',
longitude='-115.81W')
   'Coordinates: 37.24N, -115.81W'
```

```
>>> 'Coordinates: {lat}, {long}'.format( **mydict )
'Coordinates: 37.24N, -115.81W'
```

Python String Formatting

It easy to get help:

```
>>> help()
help> topics
help> FORMATTING
```

Comprehensions

- Comprehensions are an unusual structure that can be very important to understanding a program.
- Comprehensions tend to look like other common programming patterns but are in fact very different.
- Look for the keyword `for`.

Comprehensions

A simple list comprehension that creates a list of 0..9:

```
>>> [ i for i in xrange( 0, 10 ) ]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- The square brackets indicate a list
- The `for` keyword indicates a comprehension
- The `xrange( 0, 10 )` generates the data
- The variable after the `for` indicates where each value from the data is stored
- And the name immediately after the bracket receives each value generated.

Comprehensions

```
>>> [ 0 for i in xrange( 0, 10 ) ]             # The output is a constant
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> [ i**2 for i in xrange( 0, 10 ) ]      # The output can be an expression
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Note that a comprehension is an expression. Which means that comprehensions can be nested. Woo hoo! Let’s pause.

Comprehensions: Some Examples

Simple way to initialize a 2D matrix. Rows by default will have indexes 0,1,2, and the values on each row will be 1,2,3.

```
>>> a=[[b for b in xrange(1,4) ] for a in xrange(0,3)]
>>> a
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]
>>> a[2][2]
3
```

A little trickier:

```
>>> a=[[b+(a*10) for b in xrange(0,3) ] for a in xrange(0,3)]
>>> a
[[0, 1, 2], [10, 11, 12], [20, 21, 22]]
>>> a[2][2]
22
```

Comprehensions: Some Examples

A `dict` comprehension:

```
>>> a={ v: l for (l,v) in enumerate( 'The letters are keys' ) }
>>> a
{'a': 12, ' ': 15, 'e': 17, 'h': 1, 'k': 16, 'l': 4, 's': 19, 'r': 13, 't': 7, 'y':
18, 'T': 0}
```

- Note that the key `T` has the value 0, and `e`, 17 because of the `e` near the end of the string.
- Dict comprehensions are easy to spot because of the curly braces, the value with a ‘`:`’ and the `for` keyword.

Comprehensions: Some Examples

A `set` comprehension:

```
>>> a={ l for l in 'The letters are keys' }
>>> a
set(['a', ' ', 'e', 'h', 'k', 'l', 's', 'r', 't', 'y', 'T'])
```

- Set comprehensions are easy to spot because of the curly braces and the `for` keyword, but no colon where the value is.
- The “`:`” is what differentiates a `dict` comprehension from a `set` comprehension.

Comprehensions: Some Examples

A Generator Comprehension (wow!):

```
>>> a=( math.cos( b/10.0 ) for b in xrange( 0, 31 ) )
>>> for b in a:
...     print( b )
...
1.0
0.995004165278
0.980066577841
...
-0.904072142017
-0.942222340669
-0.97095816515
-0.9899924966
```

The important thing to notice is that the *Generator Comprehension* can be assigned to a variable and then processed in the `for` loop. There could be different assignments to “`a`” and then one `for` loop. One generator could use `cos` and one `sin`, for instance.

Comprehensions: No More Cryptic Examples!

The take away here is that if you come across this pattern in code you need to understand, *Google* `Python Comprehensions` and read up on the subject.

I originally had trouble really understanding comprehensions - especially nested comprehensions. But after I wrote these slides, it made much more sense.

So, either these slides are really good or I paid more attention this time around!

Slices

Slices are a way of accessing parts of various data structures that support indexed access. These include:

- Lists
- Tuples
- Character strings
- Various user defined types.

Slices

Slides have the following general components:

```
<start>:<end-excluded>:<stride>
```

with each part optional, but at least one “`:`” or an integer is required for a slice. As we have seen:

```
list_name=[0,1,2,3,4]
list_name[0]
```

would access one element at the start of the list. And the following items 2,3,4.

```
list_name[2:5]
```

Slices

Slides have the following general components:

`<start>:<end-excluded>:<stride>`

“End-excluded” is a little non-obvious. However, it allows the value to take on the default, and sensible, value of “length-of-object”. So:

`a[2:] ⇒ [2, 3, 4, 5, 6, 7, 8, 9]`

which is quite useful and prevents us from writing ugly `len(a)-2` type of constructs that you have to write in some other languages.

Slices

Slides have the following general components:

`<start>:<end-excluded>:<stride>`

“Stride” means how many elements to move to obtain the next item in the object.

- A positive stride (>0) means to move to increasing indexes (“To the right”).
- A negative stride means to move to decreasing indexes (“To the left”).

Slices

A slice of an object is itself a new object.

The default for `<name>[:]` is the entire object. Therefore a statement like

```
newlist = thelist[:]
```

is a handy way to make a quick, shallow, copy of an object that support slices.

Slices

```
>>> a=[ i**2 for i in xrange( 0, 10 ) ]
>>> b=a
>>> a is b
True
>>> b=a[:]
>>> a is b               # Shows that objects associated with a and b different.
False
```

Slices

Slices have the following general components:

`<start>:<end-excluded>:<stride>`

`Start`:

- If positive, index, 0 based from start of object. If negative, one based from end of object.

  `a[0] ⇒ 0, a[-1] ⇒ 9`

- Default: *first item* in object.

Slices

Slices have the following general components:

`<start>:<end-excluded>:<stride>`

`End-excluded`: Item to left of indexed element

- Default for `end-excluded`: *To the end*.

```
a[0:2] ⇒ [0, 1]
a[0:] ⇒ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a[-1:] ⇒ [9]
a[-1::-1] ⇒ [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Slices

Slides have the following general components:

`<start>:<end-excluded>:<stride>`

`Stride`: Items to skip. Negative stride lists items in reverse order.

- Default for `stride`: 1.

```
a[0::1] ⇒ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
a[-1:] ⇒ [9]
a[-1::-1] ⇒ [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
a[-1:0:-1] ⇒ [9, 8, 7, 6, 5, 4, 3, 2, 1]
a[-1:-1:-1] ⇒ [] # huh
```

Slices

Slides have the following general components:

```
<start>:<end-excluded>:<stride>
```

Slices are objects! (Remember when I said “Python is Objects”?)

```
s=slice( -1,None,-1 )
a[s] ⇒ [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Interesting.

Slices

```
def p_list( list, forward=True ):
    if forward:
        s = slice( 0, None, None )
    else:
        s = slice( -1, None, -1 )
    print( list[s] )

l = list( xrange( 10 ) )
p_list(l)
p_list(l, forward=False )
```

```
doc => python a.py
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Slices

It’s not too hard to imagine a data structure where you might wish to sample items at different rates.

You could define an accessor slice, and change it as you wish to access data at a finer sample sizes from the full list.

Import

Import associates one or more names with a Python code object that is read from a file.

There are standard modules included with Python:

```
import os
import argparse
```

Import

Import associates one or more names with a Python code object.

But, beware:

```
>>> import numpy as np
>>> np
>>> <module 'numpy' from '/opt/local/Library/Frameworks/.../site-
packages/numpy/__init__.pyc'>
>>> np=0
>>> np
>>> 0
>>> import numpy as np
>>> np
>>> <module 'numpy' from '/opt/local/Library/Frameworks/.../site-
packages/numpy/__init__.pyc'>
```

Import

Import associates one or more names with a Python code object.

So, remember that import is just a fancy assignment statement for names and object modules. If you reuse the name that you specified in the import, you will lose access to the module so long as that name is in *scope*.

Import

There are many variants of the import statement. See `>>> help( ‘import’ )` in Python or see the tutorials.

Generally avoid use of `as` except when it is obvious. For instance:

```
import numpy as np
import copy.copy as copy
import copy.deepcopy as deepcopy
import annoyinglongname as aln
```

all make sense to me. But, if I use the name infrequently, I do not use `“as”`.

Import

A module is only actually imported once into a program unless special steps are taken. Sample module:

```
def p_list( list, forward=True ):
    if forward:
        s = slice( 0, None, None )
    else:
        s = slice( -1, None, -1 )
    print( list[s] )

print( 'Imported!' )
```

On import, the def and the print are executed.

Import

Importing the module:

```
>>> import a
Imported!
>>> a
<module 'a' from 'a.pyc'>
>>> a=0
>>> a
0
>>> import a
>>> a
<module 'a' from 'a.pyc'>
```

Note that the module level statements only ran once (“Imported” printed once). The second import statement did reassign ‘a’ to the module code object, but it did not re-import it.

Import

- Avoid `from <module> import *`

  This imports all names into the current set of names (the current *Namespace*), without any indication of which module the names came from. This can be very confusing, and could cause names you defined to be redefined.

Import

- `from <module> import <name>[,name...]`

  This form is useful for importing “standard” names into your program.

  ```
  from copy import copy, deepcopy
  ```

  seems fine to me.

Import

Avoid using names from an imported module that start with underscore (_)

By convention, such names are private to the module and should not be used by module importers.

Modules

A module is a python source file that is `import`-ed into another python module.

The top level statements in the module are evaluated at import time. Typically, there might some assignments at the start of the module followed by some number of `import` statements, and function `def`s.

Recall that unless special steps are taken, a module is actually only imported once per program run. So, statements in the module will only execute once.

This means you can safely initialize a module once, no matter how many times it is imported.

Importing Your Own Modules

The import statement does not use file names. The conversion of the module name to a specific filename is operating system specific.

- Generally, for module `mymodule`, python will search the current module search path for a file named `mymodule.py`.
- Module names with dots in them are converted to directory names. A search for `my_app.mymodule` will search the current module search path for a directory named `my_app` that contains the file `mymodule.py`.

Importing Your Own Modules

The current module search path includes builtin modules, currently loaded modules and the directories listed as follows:

```
>>> import sys
>>> print sys.path
['', '/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python27.
zip', …,'/Library/Python/2.7/site-packages']
```

Importing Your Own Modules

It can be very convenient to make a directory of modules. For example:

> Your main script is in the top level directory (`~/bin`) and the package a subdirectory immediately under that directory (`~/bin/mypackage`).

The module directory (`~/bin/mypackage`) needs an `__init__.py` file. This file can be empty or it can contain statements that initialize the package.

The module `mymodule` needs to be in the file *mymodule.py* in directory `~/bin/mypackage`. Import as `import mypackage.mymodule.`

Importing Your Own Modules

The `__init__.py` file can contain statements to import the modules that are part of the package.

For instance, file *mypackage/__init__.py* could contain `import mymodule`.

Import as `import mypackage`

If `mymodule` contains `my_cool_function`, you could call it as `mypackage.mymodule.my_cool_function()` and import it as

```
from mypackage.mymodule import my_cool_function as mcf
```

Importing Your Own Modules

You can import a module and reference names using the full path, and also import a few common names from the module and alias them with the

```
from <module> import <name> as <name>
```

This will not cause the module to be imported more than once (modules are only imported once). It simply assigns new, shorter names to objects.

Importing Your Own Modules

```
>>> from mypackage.mymodule import my_cool_function as mcf
Imported
>>> mcf()
My Cool function
>>> mcf=0
>>> mcf
0
>>> import mypackage # Note that __init__.py imported mymodule
>>> mcf=mypackage.mymodule.my_cool_function
>>> mcf()
My Cool function
>>>
```

Import is pretty similar to an assignment! Exactly? I'm not sure.

Importing Your Own Modules

There are a number of ways of structuring your own modules. The simplest is as I have shown. It will work in both versions of Python.

There are relative imports and other special cases you can research if needed.

Importing Your Own Modules

```
doc => ls -alR

total 8

drwxr-xr-x    4 mike  staff   136 Oct 25 19:26 .
drwxr-xr-x@ 229 mike  staff  7786 Oct 25 19:26 ..
-rwxr-xr-x    1 mike  staff   102 Oct 25 19:26 my_prog
drwxr-xr-x    6 mike  staff   204 Oct 25 19:19 mypackage

./mypackage:
total 32
drwxr-xr-x  6 mike  staff  204 Oct 25 19:19 .
drwxr-xr-x  4 mike  staff  136 Oct 25 19:26 ..
-rw-r--r--  1 mike  staff   16 Oct 25 19:04 __init__.py
-rw-r--r--  1 mike  staff  138 Oct 25 19:05 __init__.pyc
-rw-r--r--  1 mike  staff   77 Oct 25 19:18 mymodule.py
-rw-r--r--  1 mike  staff  278 Oct 25 19:19 mymodule.pyc
```

Importing Your Own Modules

Reference:

```
doc => cat my_prog
#!/usr/bin/python -tt

import mypackage
from mypackage.mymodule import my_cool_function as mcf

mcf()
```

Importing Your Own Modules

```
doc => cd mypackage/
doc => cat __init__.py
import mymodule

doc => cat mymodule.py
print( 'Imported' )

def my_cool_function():
    print( 'My Cool function' )
```

Testing Your Modules

```
doc => cat mymodule.py
print( 'Imported' )

def my_cool_function():
    print( 'My Cool function' )

if __name__ == '__main__':
    print( 'Testing the module' )
    my_cool_function()

doc => python mymodule.py
Imported
Testing the module
My Cool function
```

Import (An Aside on the `copy` Module)

Note: the copy module is the standard way to copy objects.

- `Copy.copy` will copy the named object, but it will not look into the object for other mutable objects to copy.
- `Copy.deepcopy` will copy the named object, and also copy all mutable objects within.

Import (An Aside on the `copy` Module)

- `copy.copy( [ {‘key’: ‘value’, ‘key2’: ‘value2’ } ] )`

  Copies the list, but not the dicts in the list.

- `copy.deepcopy( [ {‘key’: ‘value’, ‘key2’: ‘value2’ } ] )`

  Copies the list and the dicts in the list.

I personally almost always use `copy.deepcopy`. For reasons I do not really understand, it is often suggested `copy.copy` be used. `copy.deepcopy` seems safer.

Scope

We have talked about:

- Modules
- Functions
- Statement blocks
- and assigning objects to names.

What we have not talked about is where names are “visible”. Names are stored in *namespaces*. There is a hierarchy of searching through namespaces which determines the *scope* that a name is visible and which namespace will provide the name.

Scope

First, statement blocks. Statement blocks **do not** create a namespace that limits the visibility of a name. Many other languages do.

```
if test_var:
    a = 1
else:
    b = 1
```

Depending on the value of `test_var`, either `a` or `b` will be a defined name after the if statement. However, which is does not depend on scope.

Scope

Compare the previous with C:

```
if (test_var) {
  int a = 1;
} else {
  int b = 1;
}
```

Neither `a` nor `b` will ever be in scope after the `if` statement because the curly braces start a scope block. Compound statement scoping like this **does not exist** in Python.

Scope

- When a function starts to execute, a *Local* namespace is associated with it.
- Names will be searched for in the current function, any enclosing functions, and then within the module.
- The searches in the function namespaces are called the *Local* and *Enclosing* namespaces.
- The *Enclosing* namespace is a namespace associated with a function that defined the current function. Ie:

```
def func1():
    a = 1
    def func2():
        print( ‘a = %d’ % a )
```

Scope

After local and enclosing, the next namespace that is searched is the *Global* namespace.

- An *unqualified* name exists in the current module's namespace.
- A *qualified name* (eg: sys.path) provides an explicit module path to find the module's *Namespace*.
- Note that the first part of the *qualified name* must exist in the scope of namespaces that will be searched.

Using the above example, the `sys` module must be imported in the current scope in order to find `sys.path`. Since the name spaces to be searched after the initial name is explicit, those namespaces are not part of the scope per se.

Scope

- Note that the main program executing is in fact a module, named `__main__` and therefore has a global namespace.
- Names within different modules exist in different namespaces and therefore can only be accessed from outside their module using some form of an explicit module path statement. For instance, a `from … import … as` statement or a qualified name such as `mymodule.name`.

Scope

- The final namespace to be searched is the *Builtin Namespace*. `__main__` is an example name from the builtin namespace.

The scope rule is usually referred to as the LEGB rule:

- *Local*
- *Enclosing*
- *Global*
- *Builtin*

Scope

So, what namespace are names stored in when an assignment to a name is made (using `=, import, def,` etc …)?

> Names are stored in the current namespace with one or two exceptions - depending on what Python version.

Recall that the current namespace is either the *Global* (module level) or *Local* namespace. While I believe it possible to change the *Builtin* namespace, we will skip that.

Scope

To change the *Global* namespace*:*

- Define the name within the module and not within any functions.
- Or include a `global <name>` statement within the function.
- In Python 3, a `nonlocal <name>` statement is used to change an existing name in an enclosing function (*Enclosing* scope).

To change the *Local* namespace:

Define a name within a function. Use assignment, `import`, `def`, etc.

Scope - Global Names Examples

```
a = 1    # A name defined at the module (global) level
_pvt = 1 # A name defined at module (global) level that should not be
         # used outside of the module.

def myfunc():    # A function defined at the global level
    pass

def _myfunc2():  # A private function, by convention
    global a

    a = 2        # Changes the name ‘a’ at the global level

def myfunc3():
     a = 4       # Adds a local name ‘a’
```

Scope - Global Names Examples

```
a = 1    # A name defined at the module (global) level

def does_not_work():
    a = a    # I don’t understand why the right hand ‘a’ is not located in
             # global scope and saved in local.  In any case, this does
             # not work.  Hmmm… stumped myself. I guess the assignment target
             # scope is resolved first, and then that Namespace is bound to
             # all copies of the name in the statement.
             # Or perhaps within a statement, a name can only be bound to
             # one namespace?
```

Scope - Dynamic?

```
a = 'The module'

def f1():
    a = 'The function f1'
    print( 'called f1' )
    f2()

def f2():
    f3()

def f3():
    print( 'a = %s' % a )
```

continued….

```
f1()
<save> as a.py
python a.py
called f1
a = The ...
```

Scope - Dynamic?

```
a = 'The module'

def f1():
    a = 'The function f1'
    print( 'called f1' )
    f2()

def f2():
    f3()

def f3():
    print( 'a = %s' % a )
```

continued….

```
f1()
<save> as a.py
python a.py
called f1
a = The module
```

Scope - Dynamic or Enclosing?

```
a = 'The module'

def f1():
    a = 'The function f1'
    print( 'called f1' )

    def f2():
        print( 'a = %s' % a )

    f2()
```

continued….

```
f1()
<save> as b.py
python b.py
called f1
a = The ...
```

Scope - Dynamic or Enclosing?

```
a = 'The module'

def f1():
    a = 'The function f1'
    print( 'called f1' )

    def f2():
        print( 'a = %s' % a )

    f2()
```

continued….

```
f1()
<save> as b.py
python b.py
called f1
a = The function f1
```

Scope - A final Thought

Names are looked up at run-time. We have seen numerous examples of this.

However, *where* we look for the names is defined when the source is written.

Scope is not dynamic.

Using Classes

- Classes are user defined types.
- Objects of these types can be constructed as needed.
- To create an object of class `MyClass`:

```
new_class_obj = MyClass()
```

- Classes can be imported via the `import` statement the same way any Python code object can be imported.

Using Classes

- The class may require or accept optional arguments when it is created:

```
new_class_obj = MyClass( arg1, opt_arg )
```

- The special routine `__init__` within the class defines what arguments are required via its argument list. `__init__` is the class constructor and initializes a new instance of the class.

```
class MyClass:
    def __init__( self, arg1, opt_arg = None ):
        pass
```

Using Classes

The class will likely have various attributes that you can access. The names these attributes have act like any other Python name. They refer to an object of some type:

```
nco = MyClass(1)              # Makes an object
nco.set_something( 10 )       # Calls a class method (similar to a function)
print ‘The value: %d’ % nco.something      # Prints a class attribute
print ‘The function value: %d’ % nco.thisthing()    # Class method value
```

Using Classes

Sometimes you will need to pass a routine name as a callback. A bound class method can be used. For instance:

```
myobj = NewClass( ‘Title’ )

some.modules.rtn( xxx, callback=myobj.call_back_rtn )
```

- Note that `myobj.call_back_rtn` does not have an argument list, ie: `(...).`
- By providing just the name, we are passing a Python code object.
- If we had an argument list, we would have called `myobj.call_back_rtn` and its value would be passed.

Using Classes

You should not be surprised to learn that classes themselves are Python objects.

```
if test_val:
    c = MyClassOne
else:
    c = MyClassTwo

new_obj = c( arg )
```

`new_obj` will be either of type class `MyClassOne` or `MyClassTwo`, depending on test_val. Note that there is no argument list on the assignment to “`c`”

Using Classes

You should not be surprised to learn that classes themselves are Python objects.

```
if test_val:
    c = MyClassOne()
else:
    c = MyClassTwo()

new_obj = c( arg )
```

However, actually constructing the classes is valid if `MyClassOne` and `MyClassTwo` return class objects themselves. Classes can make classes. Whew!

Using Classes

Classes are usually composed from different base classes. This allows the class writer to reuse code from different classes in a new class. It looks something like this:

```
class B1( object ):
    def rtn_b1( self ):
        print( 'rtn_b1' )

class B2( object ):
    def rtn_b2( self ):
        print( 'rtn_b2' )

class MyClass( B1, B2 ):
    def my_rtn( self ):
        print( 'Called my_rtn' )
```

```
m = MyClass()
m.my_rtn()
m.rtn_b1()
m.rtn_b2()
dir( m )

doc => python a.py
rtn_b1
rtn_b2
Called my_rtn
['__class__', '__delattr__', ...
, 'my_rtn', 'rtn_b1', 'rtn_b2']
```

Using Classes

The important point to note is that class `MyClass` now has routines `rtn_b1` and `rtn_b2`. Functionality of these routines has been *inherited* by `MyClass`.

```
class B1( object ):
    def rtn_b1( self ):
        print( 'rtn_b1' )

class B2( object ):
    def rtn_b2( self ):
        print( 'rtn_b2' )

class MyClass( B1, B2 ):
    def my_rtn( self ):
        print( 'Called my_rtn' )
```

```
m = MyClass()
m.my_rtn()
m.rtn_b1()
m.rtn_b2()
dir( m )

doc => python a.py
rtn_b1
rtn_b2
Called my_rtn
['__class__', '__delattr__', ...
, 'my_rtn', 'rtn_b1', 'rtn_b2']
```

Using Classes

For class writers, classes are a great way to reuse or customize pieces of code without repeating the code.

For class users, classes are a great way to customize bits of a library.

I will have to refer you to the library of classes to see exactly what can be customized.

Exceptions

An *exception* has occurred when the normal flow of the program is interrupted.

Exceptions are caused by:

- Errors in program execution
- Program detected logic errors

Note: I added this section at the last moment because I thought it might be helpful. Please bear with me if there are errors! (There was an error in the previous sentence. But, this is all about errors in code, so let's just handle it and move on!)

Exceptions

A simple example:

```
a = 0
b = a/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or
modulo by zero
```

Obviously, Python can not divide by zero, so the `ZeroDivisionError` is *raised*.

Exceptions

In Python, exceptions can be *caught* by the `try`… `except` statement.

```
>>> try:
...     b=1/0
... except ZeroDivisionError:
...     print 'Zero divide, setting b to 0.'
...     b = 0
...
Zero divide, setting b to 0.
>>>
```

Here, the divide by zero error was `handled`, a message output, and a suitable fix for the application applied.

Exceptions

Sometimes, we just want to output an error message, but still want the error raised:

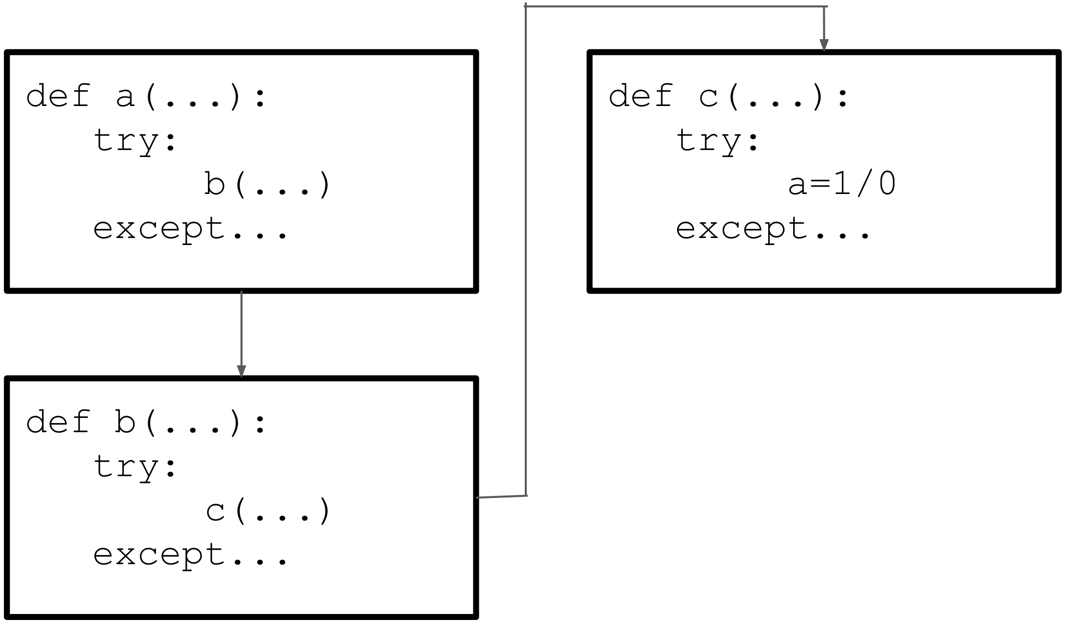
```
>>> try:
...     b=1/0
... except ZeroDivisionError:
...     print '*** Zero divide, setting b to 0.'
...     raise
...
*** Zero divide, setting b to 0.
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: integer division or modulo by
zero
```

Exceptions

- If an exception occurs, each try statement that was executing is checked for any except statements that could handle the error.
- If no except statements match the error, then the final exception handler executes which prints the traceback, the error message, and causes the program to exit.

That’s a lot to grasp, but hopefully the graphic on the next page will explain it.

Exceptions



```
def a(...):
   try:
        b(...)
   except...
```

```
def b(...):
   try:
        c(...)
   except...
```

```
def c(...):
   try:
        a=1/0
   except...
```

`except` statements for

- rtn c
- rtn b
- rtn a
- global exception

are tried in order. If an except clause matches, then the exception is said to be *handled*.

Program control picks up after the try statement that matched.

Exceptions

So, what are exception names?

- They are actually classes.
- Usually, the class is empty.
- Class inheritance is used when matching against the `except` statements.
- This means you can make your own exception types if you wish.
- https://docs.python.org/2/library/exceptions.html covers the builtin exceptions.

Exceptions

For instance, we have seen `ZeroDivisionError`. If we read the Python documentation, we will see that `ArithmeticError` is a base class for `ZeroDivisionError` as well as several other errors.

This means you could write:

```
try:
    a=1/0
except ArithmeticError:
    print( ‘Error detected’ )
```

This except `ArithmeticError` would catch `ZeroDivisionError`, `OverflowError`, etc. because `ArithmeticError` is the base class.

Exceptions

These are builtin to Python, but if you saw the code, it would look roughly like:

```
class ArithmeticError: pass
class OverflowError( ArithmeticError ): pass
class ZeroDivisionError( ArithmeticError ): pass
class FloatingPointError( ArithmeticError ): pass
```

Exceptions

Avoid the following:

```
try:
    <your code>
except:
    <error handling code>
```

Such code will catch all errors in <your code> and will tend to hide many errors that you should know about.

Exceptions

And finally:

```
try:
    a=1/0
except ZeroDivisionError as e:
    print( e )
```

will print ‘`integer division or modulo by zero`’. The `as` syntax gives you access to the error message associated with the exception.

Thanks!

Questions? Comments?

There are so many more topics we could cover…!

Just try to remember the basic rules of Python and when something new comes along, try it on your favorite box and see what happens!

Michael Porter

10/26/2015 - Python 2.7 with a wee bit of version 3 mentioned.