

# Learning Python

## Getting results for beamlines and scientific programming

Intermediate python: Intro to classes

# Outline of topics to be covered

1. What is an object
2. Example: defining a simple class
3. Objects vs. Modules
4. Aside: Iterators
5. More features of objects
  - inheritance
  - polymorphism
  - operator overloading





# What is an object? OOP for the uninitiated.

Object-oriented programming is considered the “latest and greatest” in the computer science game.

Objects combine data and the associated software for working with that data into a single unit.

Objects have other advantages for code design, but we will cover that later.



# Objects are not as exotic as they seem: you have already used python objects

- Example: when you created lists or strings, they automatically came with a set of functions for use with them:

```
>>> a
[1, 2, 3, 5, 7, 11]
>>> a.append(13)
>>> a.insert(0,17)
```

```
>>> s = "this is a string"
>>> s.upper()
'THIS IS A STRING'
```

- Strings and lists also come with the plus (+) operator defined.

```
>>> [1,2,3] + [4,5,6]
[1, 2, 3, 4, 5, 6]
```

```
>>> s.capitalize() + ". " + "And another one."
'This is a string. And another one.'
```



# Example: defining a simple class

```
class MyClass(object):
    "Document the class here"

    def __init__(self, initvar):
        self.var = initvar

    def MyRoutine(self, var):
        '''Doc for MyRoutine here'''
        print 'self.var =', self.var
```

```
>>> import democlass
>>> myobj = democlass.MyClass(1)
>>> myobj.MyRoutine('?')
self.var = 1
>>> myobj.var
1
>>> myobj.var = 2
>>> myobj.var
2
```

## Define an object (called MyClass)

- defines a new object type
- based on existing class "object"

**Define a method (function):** `__init__` is called when the object is created, parameter is saved as `var`

## Define a method to do something:

MyRoutine. Notes:

- 1<sup>st</sup> param (self) points object
- var is local and is not used

## Now use the class

### Create the object

- Calls `__init__`. Note that self is taken from object name.

### Calling the method

- Note: variables in objects can be accessed externally (no public vs. private members)



## Example: defining multiple instances

```
class MyClass(object):
    "Document the class here"

    def __init__(self, initvar):
        self.var = initvar

    def MyRoutine(self, var):
        '''Doc for MyRoutine here'''
        print 'self.var =',self.var
```

```
>>> import democlass
>>> myobj1 = democlass.MyClass(1)
>>> myobj1.MyRoutine('?')
self.var = 1
>>> myobj2 = democlass.MyClass(2)
>>> myobj2.MyRoutine('?')
self.var = 2
>>> myobj1.var
1
>>> myobj2.var
2
```

Here we define two copies of our class (myobj1 and myobj2).

Note the data in each object is defined independently.



# Objects (classes) vs. Modules

One can place both data and associated functions in a module or in a class. They do sort of similar things. What is the difference?

```
var = None
def MyRoutine():
    print 'var =', var
```

```
>>> import demomod as dm
>>> dm.var = 3
>>> dm.MyRoutine()
var = 3
```

```
class MyClass(object):
    def __init__(self, initvar):
        self.var = initvar

    def MyRoutine(self):
        print 'self.var =', self.var
```

```
>>> import democlass as dc
>>> ex = dc.MyClass(3)
>>> ex.var
3
>>> ex.var = 4
>>> ex.MyRoutine()
self.var = 4
```

- You can only have one copy of a module in an application. Every module in your application will access the same data items.
- One can define as many copies of an object as you might want.



# Python 2 vs 3 differences

- In Python 2.x one can define an object that is not based on another class:

```
class MyClass():  
    def __init__(self, initvar):  
        self.var = initvar
```

- In Python 3+ a slightly different syntax can be used to omit the parent. However, this code means something very different in Python 2.x (so don't use it):

```
class MyClass:  
    def __init__(self, initvar):  
        self.var = initvar
```

- Python 3+ will not allow the syntax where nothing is in the parentheses (as at the top). However, if one defines the parent class as object, the same syntax can be used for both 2.x and 3+:

```
class MyClass(object):  
    def __init__(self, initvar):  
        self.var = initvar
```

*Moral of story: always define classes using `class <name>(object)` or with a parent class and you will not need to worry what version of Python you are using.*

The Advanced Photon Source is an Office of Science User Facility operated for the U.S. Department of Energy Office of Science by Argonne National Laboratory



# Reminder: dir shows the variables and routines associated with a class

```
class MyClass(object):
    def __init__(self, initvar):
        self.var = initvar

    def MyRoutine(self):
        print 'self.var =',self.var
```

Note that the functions we defined (`__init__` and `MyClass`) are listed, plus the variable (`var`)

- Python provides many other special variables and functions (with names beginning and ending with two underscores [`__`]), we will talk about a few of them in just a minute.

```
>>> import democlass as dc
>>> ex1 = dc.MyClass(1)
>>> dir(ex1)
['MyRoutine', '__class__',
 '__delattr__', '__dict__', '__doc__',
 '__format__', '__getattr__',
 '__hash__', '__init__', '__module__',
 '__new__', '__reduce__',
 '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__',
 'var']
>>>
>>> ex1.__class__
<class 'democlass.MyClass'>
>>> ex1.__str__()
'<democlass.MyClass object at 0x37e210>'
>>> ex1.__module__
'democlass'
```



# Detour: Introducing iterators

What is an iterator?

- When we want to work sequentially with a large amount of data, we may not want to have to have to read/compute all of the data items before starting to process them. Why?
  - We may need to process the data items only until we reach a particular item
  - The amount of memory needed to store all the items may be excessive
- Where might one want to use an iterator?
  - reading in a large file
  - computation of a large sequence of values, particularly if we will stop when some condition is matched or each value is only used one time.



## for loops use iterators (if defined)

In these examples, Python loop generates each value for the “for” loop, values are generated only as they are used. This is the central concept in iterators

```
for myitem in range(10000): pass
```

```
fp = open('myfile', 'r')  
for myline in fp: pass
```

```
mydict = {1:'a', 2:'b', ...}  
for mykey in mydict: pass
```



# Implementing an iterator

- Implementing an iterator requires addition of two methods to a class: `__iter__` and `next`:
- `__iter__` is called with no arguments and returns an object that contains the next function (usually the name of the class containing `__iter__`, but on occasion a new class can be created for iteration)
- `next` is also called with no arguments and either:
  - returns the next value in the sequence or
  - raises a `StopIteration` exception when there are no more values to return.

For more on iterators see <http://docs.python.org/library/stdtypes.html#typeiter>



# Simple Iterator example

```
import math
class sqrtrange(object):
    def __init__(self, stop):
        print "calling __init__"
        self.stop = stop
    def __iter__(self):
        print "calling __iter__"
        self.count = 0
        return self
    def next(self):
        if self.count >= self.stop:
            raise StopIteration
        res = math.sqrt(self.count)
        print "calling next"
        self.count += 1
        return res

slist = sqrtrange(1000)
for sq in slist:
    print sq
    if sq >= 3: break
```

This defines a simple class for a “list” of square roots .

- `__init__` is called when the object is created.
- `__iter__` is called when the iteration is started
- `next` is called to obtain each item

```
toby$ python iter_ex.py
calling __init__
calling __iter__
calling next
0.0
calling next
1.0
calling next
1.41421356237
...
(five iterations skipped)
...
calling next
2.82842712475
calling next
3.0
toby$
```

*Note that even though we called `sqrtrange(1000)` the code only computed the square-root values for the first 10 values in the sequence. If this had not been implemented as an iterator, class would have to compute the square-root of all 1000 values.*



# Adding a bit more class to our class

We can add more capability to our class

- By adding `__len__` then we can use `len(object)`
- By adding `__getitem__` we can index individual items in the “list”

```
import math
class sqrtrange(object):
    def __init__(self, stop):
        self.stop = stop
    def __iter__(self):
        self.count = 0
        return self
    def next(self):
        if self.count >= self.stop: raise StopIteration
        result = math.sqrt(self.count)
        self.count += 1
        return result
    def __len__(self):
        return self.stop
    def __getitem__(self, i):
        if i > self.stop:
            raise IndexError, 'index out of range'
        return math.sqrt(i)
```

*In this example we still only compute square-root items as they are needed but simulate a large list.*

```
toby$ python
>>> import iter_ex1
>>> sl = iter_ex.sqrtrange(10000)
>>> len(sl)
10000
>>> sl[16]
4.0
>>> sl[500000]
IndexError: index out of range
>>>
```