Introduction to Python
Class notes

- Examples

http://www.hpc.temple.edu/owlsnest/training
Built-in Types

- Numeric Types
  - Built-in functions
    - int()
    - long()
    - float()
    - complex()
Division

- In Python 2.x performs floor (integer) division if both operands are of type `int`
- True division if any of the operands are `float`
- In Python 3.x `/ always does true division`
- `//` operator in Python 3.x and 2.x `always` performs floor division
- Python 3.x behaviour can be enabled in 2.x
>>> from __future__ import division
>>> 1/2
0.5
>>> 1//2.0
0.0
Built-in Types

- Numeric types
  - Are immutable types
  - Value cannot be changed once created
  - Reassignment leads to creation of new object in memory
Built-in Types

- Sequence Types: Immutable
  - String (type `str`)
    - 'Hello, World'
  - Tuples (type `tuple`)
    - 1.2, 3, 'word'

- Sequence Types: Mutable
  - Lists (type `list`)
    - ['hello', 12, 23.5]
  - Sets (type `set`)
    - `set([1, 2, 3, 'string'])`
Built-in Types

- Dictionary
  - `{key1:value1, key2:value2...}
  -{"home":"988988988", "office":"0803332222", "integer":25}`
Operations

- Sequence types
  - Membership testing: `in` and `not in`
  - Concatenation: `+` operator
  - Repetition: `*` operator
  - Indexing: `a[i]`
  - Slicing: `a[i:j]`, `a[i:j:k]`

- Dictionary type
  - Indexing: `a[key]`
  - Membership testing of keys
if myint % 2 == 0:
    print "Multiple of 2"
elif myint % 3 == 0:
    print "Multiple of 3"
elif myint % 4 == 0:
    print "Multiple of 4"
else:
    print "Not multiple of 2, 3 or 4"
for i in range(5):
    print "Current value of i is ", i
    print "twice that is", i*2
List Comprehensions

```python
powertwo = [2**i for i in range(10)]
print powertwo

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```
Functions

def subst(fchar, tchar, srcstr):
    result = ""
    for char in srcstr:
        if char == fchar:
            result += tchar
        else:
            result += char
    return result

- Namespaces
Python scripts

- Similar to bash scripts
- First line
  - `#!/usr/bin/env python`
  Make executable with
  - `chmod +x scriptfile.py`
#!/usr/bin/env python

import sys

count = 0
for line in sys.stdin:
    for word in line.lower().translate(None, ',.,()"\').split():
        if word == sys.argv[1]:
            count += 1

print count
# Function to calculate quadratic roots

```python
#!/usr/bin/env python
import math

def quadroot(a, b, c):
    discr = (b * b) - (4 * a * c)
    discr_sqrt = math.sqrt(discr)
    root1 = (-b + discr_sqrt) / (2 * a)
    root2 = (-b - discr_sqrt) / (2 * a)
    return root1, root2

r1, r2 = quadroot(1, 3, -10)
print r1, r2
```
Modules

- Script with a function to calculate quadratic roots

$ ./quadroot.py

2.0 -5.0
Creating Modules

- Use our `quadroot.py` script as a module
- Can be reused by other scripts simply by importing
- We create script `callquad.py` that imports `quadroot.py`

```python
#!/usr/bin/env python
import quadroot

a, b = quadroot.quadroot(5, 3, -10)
print a, b
```
Creating Modules

- Output of callquad.py
- Also prints output of quadrooot.py script

```
$ ./callquad.py
2.0  -5.0
1.14568322948  -1.74568322948
```
Creating Modules

- `import` executes code in the modules
- Within a module `__name__` is defined
- We can execute the main block conditionally
- `__name__` is the string '__main__' when executing the main block
#!/usr/bin/env python

import math

def quadroot(a, b, c):
    discr = (b * b) - (4 * a * c)
    discr_sqrt = math.sqrt(discr)
    root1 = (-b + discr_sqrt) / (2 * a)
    root2 = (-b - discr_sqrt) / (2 * a)
    return root1, root2

if __name__ == '__main__':
    r1, r2 = quadroot(1, 3, -10)
    print r1, r2
Creating Modules

- Output of `quadroot.py` and `callquad.py`

$ ./quadroot.py
2.0 -5.0

$ ./callquad.py
1.14568322948 -1.74568322948
import...as Statement

- **import** creates a new namespace with the name of the module by default
- **import...as** can import a module into an alternate namespace
Creating Modules

...  
r1, r2 = quadroot(7, 3, 10)  
...  
- Terminates with exception

Traceback (most recent call last):
  File "./quadroot.py", line 12, in <module>
    r1, r2 = quadroot(7, 3, 10)
  File "./quadroot.py", line 7, in quadroot
    discr_sqrt = math.sqrt(discr)
ValueError: math domain error
import...as Statement

- `cmath` module has complex version of `sqrt` and other functions
- We would have to change all occurrences of `math.sqrt()` to `cmath.sqrt()`.
- Or we can import `cmath` functions into a namespace named `math`
#!/usr/bin/env python

import cmath as math

def quadroot(a, b, c):
    discr = (b * b) - (4 * a * c)
    discr_sqrt = math.sqrt(discr)
    root1 = (-b + discr_sqrt) / (2 * a)
    root2 = (-b - discr_sqrt) / (2 * a)
    return root1, root2

r1, r2 = quadroot(1, 3, 10)
print r1, r2
Creating Modules

- Output of `quadroot.py`

```bash
$ ./quadroot.py
(-1.5+2.78388218142j) (-1.5-2.78388218142j)
```
import math
import cmath
math.sqrt = cmath.sqrt

def quadroot(a, b, c):
    discr = (b * b) - (4 * a * c)
    discr_sqrt = math.sqrt(discr)
    root1 = (-b + discr_sqrt) / (2 * a)
    root2 = (-b - discr_sqrt) / (2 * a)
    return root1, root2
File input/output

- `open()` Built-in function opens a file for reading/writing and returns a `file` type
- Several methods defined including `read()` and `write()`
- `file` is an iterator type
#!/usr/bin/env python

import sys

ifile = open(sys.argv[1], 'r')
for line in ifile:
    cols = line.split()
    if "C" in cols[0]:
        print line,
ifile.close()
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>3.102166</td>
<td>11.5549</td>
<td>0.0000</td>
</tr>
<tr>
<td>C</td>
<td>4.343029</td>
<td>10.8749</td>
<td>0.0000</td>
</tr>
<tr>
<td>C</td>
<td>4.343243</td>
<td>9.41218</td>
<td>0.0000</td>
</tr>
<tr>
<td>C</td>
<td>3.102143</td>
<td>8.71322</td>
<td>0.0000</td>
</tr>
<tr>
<td>B</td>
<td>3.100137</td>
<td>7.30638</td>
<td>0.0000</td>
</tr>
<tr>
<td>N</td>
<td>4.341568</td>
<td>6.57610</td>
<td>0.0000</td>
</tr>
<tr>
<td>B</td>
<td>4.345228</td>
<td>5.13343</td>
<td>0.0000</td>
</tr>
<tr>
<td>N</td>
<td>3.103911</td>
<td>4.39795</td>
<td>0.0000</td>
</tr>
<tr>
<td>B</td>
<td>3.100340</td>
<td>2.95305</td>
<td>0.0000</td>
</tr>
<tr>
<td>N</td>
<td>4.341533</td>
<td>2.21948</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
$ ./file_example.py comfilens.txt

<table>
<thead>
<tr>
<th>C</th>
<th>3.102166</th>
<th>11.5549</th>
<th>0.0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>4.343029</td>
<td>10.8749</td>
<td>0.0000</td>
</tr>
<tr>
<td>C</td>
<td>4.343243</td>
<td>9.41218</td>
<td>0.0000</td>
</tr>
<tr>
<td>C</td>
<td>3.102143</td>
<td>8.71322</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
Exceptions

- When `math.sqrt()` is called with a negative argument we got

```
Traceback (most recent call last):
 File "./quadroot.py", line 12, in <module>
   r1, r2 = quadroot(7, 3, 10)
 File "./quadroot.py", line 7, in quadroot
   descr_sqrt = math.sqrt(descr)
ValueError: math domain error
```
Exception Handling

- Code that is executed when an exception is raised
- Exceptions are propagated up the call chain
  - Causes an abort if no handler is found
- Exception handling in Python is with the `try...except...else` construct
Exception Handling

- `try...except...else` construct

```python
try:
    
except ZeroDivisionError:
    
except ValueError:
    
else:
    ```
Exception Handling

#!/usr/bin/env python

def quadroot(a, b, c):
    discr = a**2 - 4*a*c
    discr_sqrt = math.sqrt(discr)

if __name__ == '__main__':
    try:
        r1, r2 = quadroot(1, 3, 10)
    except ValueError:
        print "Error: Negative discriminant!"
    else:
        print r1, r2
Python classes allow us to define new data types

Classes, attributes and methods, are central to the Python data model

Illustrate this with a simple example

- We create a class called `Point2d`
- Classes are created using the `class` keyword
- `class` begins a new namespace/scope
Define a class. Interactive interpreter

```python
>>> class Point2d(object):
...     x = 2.0
...     y = 3.0
...     ...

>>>```

Classes

- Attribute access

```python
>>> print Point2d.x
2.0
>>> print Point2d.y
3.0
```
Classes

- Class instantiation

```python
>>> point1 = Point2d()
>>> point2 = Point2d()
```

- Instances have their own namespace

```python
>>> point1.x = 'newstring'
>>> print point1.x
newstring

>>> print point2.x
2.0
```
Classes

- Methods
  - Are functions defined within the class scope/namespace
  - Class methods are unbound
  - Instance methods are bound
  - Bound methods have implicit first argument
Classes

- Method

```python
>>> class Point2d(object):
...     x = 2.0
...     y = 3.0
...     def printxy(self):
...         print self.x
...         print self.y
...```
Classes

- Create two class instances

```python
>>> point1 = Point2d()
>>> point2 = Point2d()

>>> point1.x = "newstring"
```
Classes

- Calling a bound method

```python
>>> point1.printxy()
newstring
3.0
>>> point2.printxy()
2.0
3.0
```
Classes

- Calling an unbound method

```python
>>> Point2d.printxy(point1)
newstring
3.0
3.0

>>> Point2d.printxy(point2)
2.0
3.0
3.0
```
Special method `__init__()`

- Defined in the same way as other methods
- But, invoked by Python when an instance is created
- Usually used with parameters to initialize an instance object
Special method `__init__()`

- `__init__()` with parameters

```python
>>> class Point2d(object):
...     def __init__(self, xval, yval):
...         self.x = xval
...         self.y = yval
...     ...
```
Special method `__init__()`

- Instantiation with arguments

```python
>>> point1 = Point2d(33.3, 0.0)
>>> point2 = Point2d(-3.0, -1.0)

>>> print point1.x, point1.y
33.3 0.0

>>> print point2.x, point2.y
-3.0 -1.0
```
Example

- **StopWatch class**
  - Measures elapsed times
- **Implement methods**
  - `start()`
  - `stop()`
  - `split()`
  - `reset()`
  - `gettime()`
Example

- **StopWatch class**
  - Measures elapsed times

- Implement behavior
  - Work with the `print` statement
  - Should iterate over stored split times when used in a `for` clause
  - Addition of two `StopWatch` types (+ operator)

- In the next slides we implement these methods and progressively add functionality
def __init__(self, hours=0, minutes=0, seconds=0.0):
    self._seconds = hours*3600 + minutes*60 + seconds
    self._running = False
__init__() Method

- Takes arguments hours, minutes, seconds
  - Takes default parameter values if not provided

```python
>>> import stopwatch as sw
>>> a = sw.StopWatch()

>>> print a._seconds
0.0

>>> a = sw.StopWatch(1, 10, 3.3)

>>> print a._seconds
4203.3
```
__init__() Method

- Keyword arguments can be used

```python
>>> a = sw.StopWatch(1, 10)

>>> print a._seconds
4200.0

>>> a = sw.StopWatch(1, seconds=10)

>>> print a._seconds
3610
```
start(), stop() Methods

- StopWatch works by recording the current time at start() and stop()
- Stores the difference in _seconds
import time

class StopWatch(object):
    ...

    def start(self):
        self._starttime = time.time()
        self._running = True

    def stop(self):
        if self._running is True:
            curtime = time.time()
            self._seconds += curtime - self._starttime
            self._running = False
Testing

- Import our class as a module

```python
>>> import stopwatch as sw
>>> import time

>>> a = sw.StopWatch()

>>> a.start();time.sleep(3);a.stop()

>>> print a._seconds
3.00312399864
```
Exception Class

```python
import time

class StopWatch(object):
    ...

    def start(self):
        self._starttime = time.time()
        self._running = True

    def stop(self):
        if self._running is True:
            curtime = time.time()
            self._seconds += curtime - self._starttime
            self._running = False
```
Exceptions

- Exceptions are also classes
- We create a new exception type
  - SWInvalidOpError
- Simplest way to create a new exception is to create an empty class that *inherits* from the `Exception` type.
class SWInvalidOpError(Exception):
    pass

class StopWatch(object):
    ...

    def stop(self):
        if self._running is True:
            curtime = time.time()
            self._seconds += curtime - self._starttime
            self._running = False
        else:
            raise SWInvalidOpError("Stopwatch not running")
reset() Method

- Resets a stopped counter to zero

```python
class Stopwatch(object):

...  

def reset(self):
    if self._running is not True:
        self._seconds = 0
    else:
        raise SWInvalidOpError("Stopwatch running")
```

...
gettime() Method

- It is an accessor method
- Builds the external interface to the type

```python
def _sec_to_hms(self, sec):
    minutes, seconds = divmod(sec, 60)
    hours, minutes = divmod(minutes, 60)
    return int(hours), int(minutes), seconds

def gettime(self):
    return self._sec_to_hms(self._seconds)
```

...
split() Method

- Appends the current elapsed time to instance variable \_splittimes
- Does not stop the counter
- Appends a StopWatch type with the current elapsed time to \_splittimes
- We keep track of how many elements in \_splittimes with variable \_num_split

**Modifications**

- \__init\__ initializes \_splittimes to []
- reset() also resets \_splittimes and \_num_split
split() Method

...  

def split(self):
    if self._running:
        curtime = time.time()
        diff = curtime - self._starttime
        splitwatch = StopWatch(*self._sec_to_hms(diff))
        self._splittimes.append(splitwatch)
        self._num_split += 1
    else:
        raise SWInvalidOpError("Stopwatch running")
...
**split() Method**

```python
def __init__(self, hours=0, minutes=0, seconds=0.0):
    self._seconds = hours*3600 + minutes*60 + seconds
    self._running = False
    self._splittimes = []
    self._num_split = 0

...

def reset(self):
    if self._running is not True:
        self._seconds = 0
        self._splittimes = []
        self._num_split = 0
    else:
        raise SWInvalidOpError("Stopwatch running")
```
Testing

```python
>>> a.start(); time.sleep(0.2); a.split();
... time.sleep(0.3); a.split(); a.stop()

>>> print a._splittimes
[<stopwatch.StOpWatch object at 0x7f32243dce90>,
 <stopwatch.StOpWatch object at 0x7f32243dced0>]

>>> print a.gettime()
(0, 0, 0.5007929801940918)
```
__str__() Method

- Currently if we use a `StopWatch` type in a print statement

```python
>>> print a
<stopwatch.StopWatch object at 0x7f8d8d8c1bad10>
```

- When we call `print` with any argument, Python implicitly calls the `__str__` method of that type
- `__str__` returns a string
- We implement the `__str__` method
___str___() Method

...  
def __str__(self):  
    hms = self._sec_to_hms(self._seconds)  
    return "{:0}:{1}:{2}".format(hms[0], hms[1], hms[2])

- format method of str type, substitutes the keywords enclosed in {} with the values in its argument list
**Iterator Protocol**

- In a `for` clause of type `for i in a`
- Python expects `a` to be a type that implements the iterator protocol
- An iterator type must implement two special methods
  - `__iter__()`
  - And, `next()`
- `__iter__()` returns an iterator type. If the type is already an iterator, it returns itself
- `next()`, returns the next item of the container, if there are no more elements, it raises the `StopIteration` exception.
- We implement `__iter__() and `next()` and modify `__init__()` and `reset()` to keep track of the next element to return
def __iter__(self):
    self._next_split = 0
    return self

def next(self):
    if self._next_split > (self._num_split - 1):
        raise StopIteration
    else:
        self._next_split += 1
    return self._splittimes[self._next_split - 1]
__iter__ and next methods

...  
def reset(self):
    if self._running is not True:
        self._seconds = 0
        self._splittimes = []
        self._num_split = 0
        self._next_split = 0
    else:
        raise SWInvalidOpError("Stopwatch running")
```python
>>> a = sw.StopWatch()
>>> a.start();time.sleep(0.3);a.split()
... time.sleep(0.2);a.split();a.split();a.stop()
>>> for i in a:
...   print i
...
0:0:0.300392150879
0:0:0.500720024109
0:0:0.500756025314
```
**__add__** Method

- The `__add__` method is called implicitly when the `+` operator is used.
- When the statement

\[
\text{op1 + op2}
\]

- The implicit call that Python makes is

\[
\text{op1.__add__(op2)}
\]
__add__ method

... 

def __add__(self, y):
tot = self._seconds + y._seconds
return StopWatch(*self._sec_to_hms(tot))
Testing

```python
>>> a = sw.StopWatch()
>>> a.start(); time.sleep(2.3); a.stop()
>>> b = sw.StopWatch()
>>> b.start(); time.sleep(1.2); b.stop()

>>> print a
0:0:2.30177211761
>>> print b
0:0:1.20131587982
>>> print a + b
0:0:3.50308799744

>>> print a * b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'StopWatch' and 'StopWatch'
```
Exercise

- Implement a Matrix class
- Support operations: addition (+), subtraction (-) and multiplication (*)