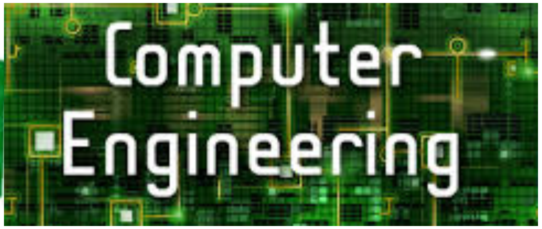
SESSION 1: INTRODUCTION TO PYTHON

Sebastien Donadio
sdonadio@uchicago.edu

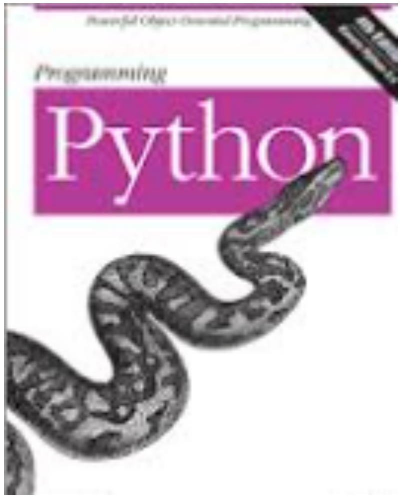
MSCA 32016– Advanced Python for Streaming Analytics

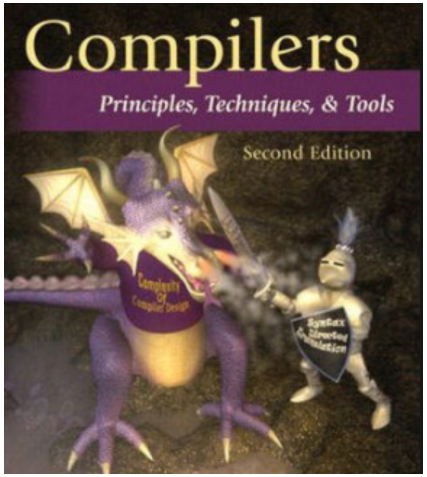
Why am I talking with you today?

- **Background**



- **Teaching**





My professional experience

Introduction

Important Facts

- Instructor:
- Sebastien Donadio sdonadio@uchicago.edu

  M 6:00p-9:00p

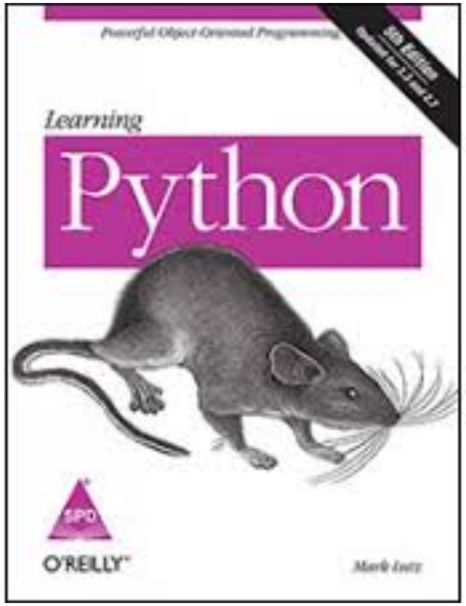
- Assistant: Maria Saenz mdcsaenz@uchicago.edu
- TA Hours on Thursday

Syllabus

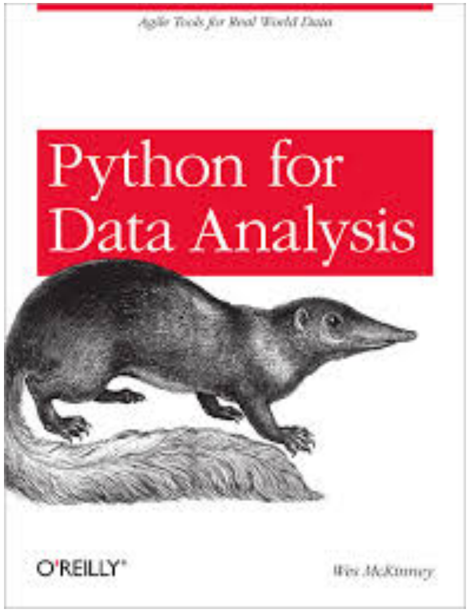
- Weekly assignments: 40%
- Course Project: 40%
- Attendance / Participation: 20%

- Late Submission
  - Prorated points will be applied according to the number of late days.
  - 1 day late: 75% of the grade will be considered.
  - 2 days late: 50% of the grade will be considered.
  - 3 days late: 25% of the grade will be considered.
  - After 3 days, no assignment will be accepted.

Books


Learning
Python
O'REILLY


Programming
Python
O'REILLY


Python for
Data Analysis
O'REILLY

Expectations

- *"I am a futures trader, and I would be interested in learning more about using python on futures data, trading models or strategies, or anything broadly finance related. Regardless"*

- *Below are some topics I am curious about and would like to cover if it's applicable to the class:*

  - *Generators --> When/why are these used? Are decorators related to this concept?*

  - *Managing virtual environments --> Are there best practices with virtual environments when building an application or collaborating with others?*

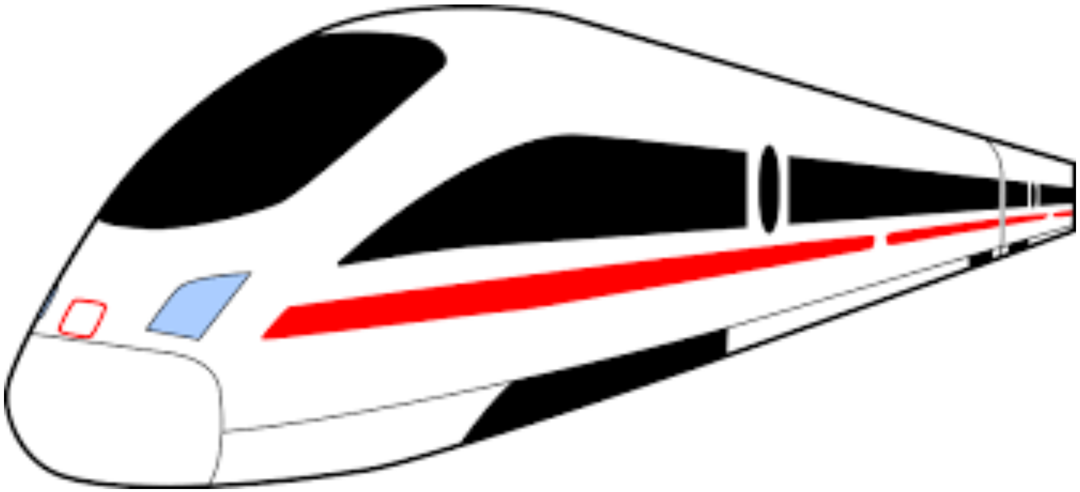
  - *Packaging code --> Best practices for packaging code so consumers can install easily, ie, pip install*

  - *Error handling --> what are the different ways to do this? eg, try/except*

  - *Lambda --> a little confused what this is when I see it (eg, filter, map reduce)*

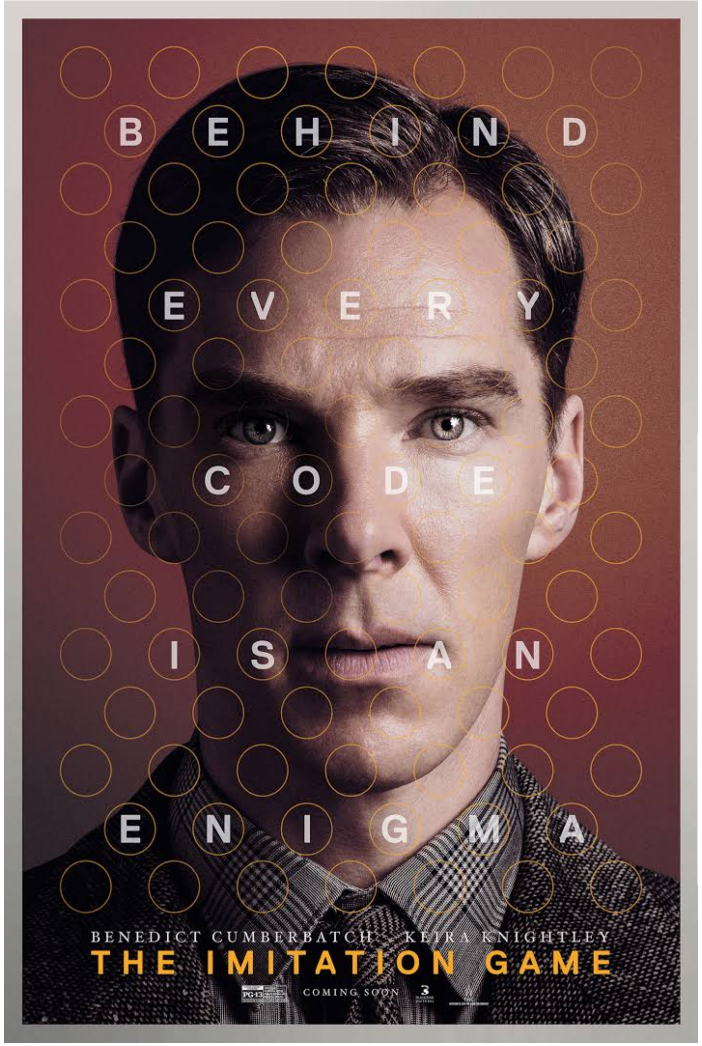
Agenda

- Python Basics with a Bullet Train speed
- More advanced Python knowledge: OOP
- Exception, Package



Brief Python Introduction

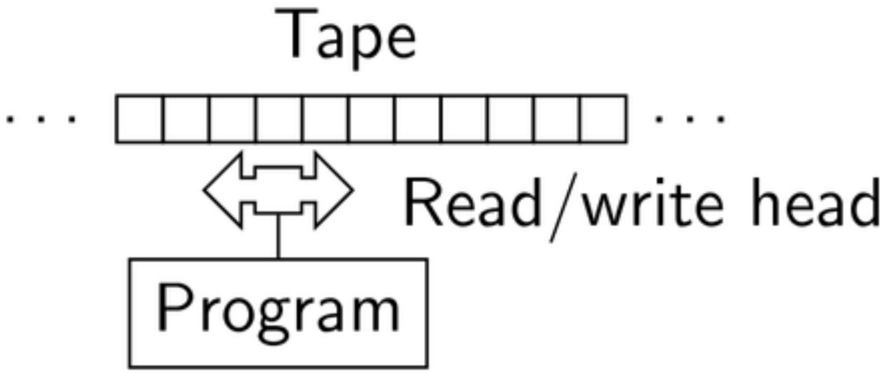
Let’s first talk about



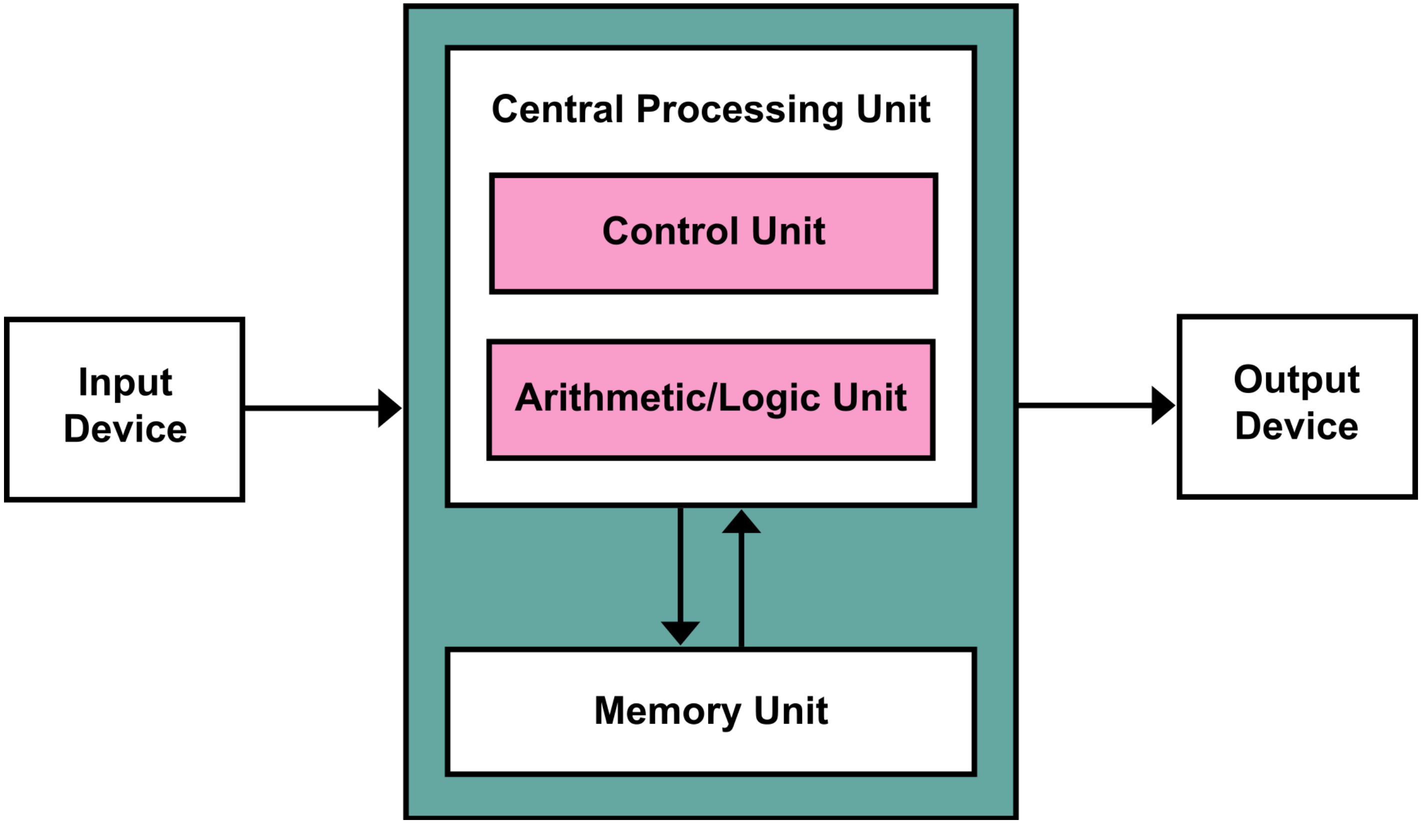
Who was this character in the movie?

Alan Turing

Turing Machine, 1936



Just after: Von Neumann architecture



Python

- Python is:
  - Widely used
  - High-level (vs low-level: assembly code, C,…)
  - General-purpose (vs DSL: HTML, Erlang,…)
  - Interpreted (vs compiled)
  - Dynamic programming language (vs static)

- Important paradigm:
  - Object-oriented
  - Imperative (vs Declarative: Erlang, Ocaml,…Specify how to do)
  - Functional/procedural
  - Dynamic type
  - Large set of libraries

Python installation

- I advise to use PyCharm or Anaconda (but you can use anything you prefer) as a IDE (Integrated Development Environment)

- Python is pre-installed with Mac OS and Linux
- For Windows, you need to install it http://python.org

```
sdonadio@SDONADIO1 ~
$ python --version
Python 2.7.8

sdonadio@SDONADIO1 ~
$ python
Python 2.7.8 (default, Jul 28 2014, 01:34:03)
[GCC 4.8.3] on cygwin
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

Running a Python program

```
#!//usr/bin/python

# A comment.
x = 12 +23

y = "Hello" # Another one.
z = 32.32

if z == 32.32 or y == "Hello":
    x = x + 1
    y = y + " World" # String concat.

print x
print y
```

```
$ python filename.py
36
Hello World
```

Indentation

- The indentation in python is not optional.
- It replaces the curly braces {}
- Use a newline to end a line of code (use \ when you want to continue the line)

- example

```
if (cond == True)
{
     print “something”
}
```

⟹

```
if cond == True:
____print “something”
```

The basics

- Assignment: =
- Comparison: ==
- Operations: +-*/% for numbers
- Logical operators: and, or, not
- Variables:
  - No declaration needed
  - No need to specify the type

Basic Data types

- Integers
  - 1 2 3 4 5
- Float
  - 1.23 343.43 43.242
- Strings
  - “foo”

Comments

- Start comments with # – the rest of line is ignored.

- You can also use

```
"""
This is a comment on many lines

"""
```

Assignment

- The assignment is done using =
- Binding a name in Python means setting a name to hold a reference to some object
- No intrinsic type
- We create a name the first time, it appears
  - `foo = 4`
- A reference will be deleted via garbage collection after any names bound to it have passed *out of scope*

Using a variable

- If you try to access a variable without creating the variable first, we will get an error

```
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Multiple Assignment

```
>>> x,y=1,2

>>> print x
1
>>> print y
2
```

Naming rules

- Case sensitive
- Don't start with a number

Reserved words:

- `and`
- `assert`
- `break`
- `class`
- `continue`
- `def`
- `del`
- `elif`
- `else`
- `except`
- `exec`
- `finally`
- `for`
- `from`
- `global`
- `if`
- `import`
- `in`
- `is`
- `lambda`
- `not`
- `or`
- `pass`
- `print`
- `raise`
- `return`
- `try`
- `while`

Understanding Reference Semantics

- Assignment manipulates references

  `x = y` (assign a reference to x)

- Assignment example:

  ```
  x = 3
  ```

  - integer 3 is created and stored in memory
  - a variable x is created
  - a reference to the memory location storing the 3 is then assigned to the name x

Understanding Reference Semantics

- Assignment manipulates references

  `x = y` (assign a reference to x)

- Assignment example:

  ```
  x = 3
  ```

  - integer 3 is created and stored in memory
  - a variable x is created
  - a reference to the memory location storing the 3 is then assigned to the name x

Mutable vs Immutable

Class	Description	Immutable?
bool	Boolean value	✓
int	integer (arbitrary magnitude)	✓
float	floating-point number	✓
list	mutable sequence of objects	
tuple	immutable sequence of objects	✓
str	character string	✓
set	unordered set of distinct objects	
frozenset	immutable form of set class	✓
dict	associative mapping (aka dictionary)	

Built-in types

- Tuples
  - Sequence of immutable Python objects
  - a=(1,2,3)
- Dictionary
  - Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces.
  - b={‘name’ : ‘foo’, ‘firstname’ : ‘las’}
- List
  - most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.
  - c=[1,23,4]

List introduction

- List is surrounded by square brackets

```
A=[1,2,4]
```

- List element can be any Python object

- A list can be empty

```
A=[]
```

- Remember:

```
for i in [1,2,3,4]:
```

- List are mutable

List (some new functions)

- We already saw:

`len(list)`

`range(…)` returning a list

- We can slice a list

`t=[1,2,3,4,5,6,7]`

`t[1:3]` → `[2,3]`

- type(t) → <class 'list'>

List (some new functions)

- `append` adding a new element
- Operator in

```
>>> 1 in [1,2,3]
True
>>> 0 in [1,2,3]
False

>>> foo
['a', 'd', 'AA']
>>> foo.sort()
>>> foo
['AA', 'a', 'd']
```

Create a list from string

- >>> foo
- ['AA', 'a', 'd']
- >>> line='I am going to split this line'
- >>> line.split()
- ['I', 'am', 'going', 'to', 'split', 'this', 'line']

- Adding many objects at the same time
- >>> foo.extend([1,2,3])
- >>> print (foo)
- ['AA', 'a', 'd', 1, 2, 3]

List and Tuples

List:

- Mutable
- l1 = [1,2]
- l1[0]=3 # ok

- id(l1) # 12345
- l1+= [4]
- Id(l1) # 12345

- C={l1: 1} # error

Tuple:

- Immutable
- t1 = (1,2)
- t1[0]=3 #error

- Id(t1) # 14345
- t1+= (4,)
- Id(t1) # 14355

- C={t1: 1} # ok

List and Dictionary

- Dictionaries are like Lists except that they use keys instead of numbers to look up values

```
>>> lst = list()
>>> lst.append(21)
>>> lst.append(183)
>>> print lst[21, 183]
>>> lst[0] = 23
>>> print lst[23, 183]
```

```
>>> ddd = dict()
>>> ddd['age'] = 21
>>> ddd['course'] = 182
>>> print ddd
{'course': 182, 'age': 21}
>>> ddd['age'] = 23
>>> print ddd
{'course': 182, 'age': 23}
```

Dictionary functions

```
>>> car={'tyre':4, 'wheel':1, 'antenna':1}
>>> print (car)
{'tyre': 4, 'wheel': 1, 'antenna': 1}

>>> car['tyre']=2
print (car)
{'tyre': 2, 'wheel': 1, 'antenna': 1}

>>> car.get('wheel')
1

>>> car.values()
dict_values([2, 1, 1])

car.keys()
dict_keys(['tyre', 'wheel', 'antenna'])
```

Dictionary functions

```
>>> for k,v in car.items():
...     print (k,v)
...
tyre 2
wheel 1
antenna 1

>>> [key for (key, value) in car.items() if value == 1]
['wheel', 'antenna']
```

➔

```
keys = []
for k,v in car.items():
    if(v == 1): keys+=[k]
print keys
```

Dictionary functions

```
>>> for k,v in car.items():
...     print (k,v)
...
tyre 2
wheel 1
antenna 1

>>> [key for (key, value) in car.items() if value == 1]
['wheel', 'antenna']

>>> del car['antenna']
>>> print (car)
{'tyre': 2, 'wheel': 1}
```

Object Oriented Programming

Object Oriented Programming in Python

- Why?
  - Better program design
  - Better modularization

Inspired by Richard P. Muller, California Institute of Technology presentation

What is an Object?

- A software item that contains variables and methods
- Object Oriented Design focuses on
  - Encapsulation:
    - dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism:
    - the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance:
    - the ability to create subclasses that contain specializations of their parents

Inspired by Richard P. Muller, California Institute of Technology presentation

Python Classes

- Python contains classes that define objects
  - Objects are instances of classes

__init__ is the default constructor

```
class point:
    def __init__(self,color,x,y,z):
        self.color = color
        self.coordinate = (x,y,z)
```

self refers to the object itself,
like *this* in Java.

Inspired by Richard P. Muller, California Institute of Technology presentation

Example: Point class

```
class Point:
    def __init__(self,color,x,y,z):
            self.color = color
            self.location = (x,y,z)
    def getColor(self):   # a class method
            return self.color
    def __repr__(self): # overloads printing
            return '%d %10.4f %10.4f %10.4f' %
                    (self.color, self.location[0],
                     self.location[1],self.location[2])

>>> point1 = point(6,0.0,1.0,2.0)
>>> print point1
6  0.0000  1.0000 2.0000
>>> point1.getColor()
6
```

Inspired by Richard P. Muller, California Institute of Technology presentation

Point class

- Overloaded the default constructor
- Defined class variables (color, location) that are persistent and local to the atom object
- Good way to manage shared memory:
  - instead of passing long lists of arguments, encapsulate some of this data into an object, and pass the object.
  - much cleaner programs result
- Overloaded the print operator

- We now want to use the point class to build a shape...

Inspired by Richard P. Muller, California Institute of Technology presentation

Shape Class

```
Class Shape:
   def __init__(self,name='Shape'):
         self.name = name
         self.pointlist = []
   def addpoint(self,el):
         self.pointlist.append(el)
   def __repr__(self):
         str = 'This is a shape named %s\n' % self.name
         str = str+'It has %d points\n' % len(self.pointlist)
         for p in self.pointlist:
               str = str + `p` + '\n'
         return str
```

Inspired by Richard P. Muller, California Institute of Technology presentation

Inheritance

```
class line(shape):
    def addcurve(self,intensity):
            self.intensity=intensity
```

- __init__, __repr__, and __addpoint__ are taken from the parent class (molecule)
- Added a new function addcurve() to add an intensity
- Another example of code reuse
  - Basic functions don't have to be retyped, just inherited
  - Less to rewrite when specifications change

Inspired by Richard P. Muller, California Institute of Technology presentation

Overloading parent functions

```
class line(shape):
    def __repr__(self):
        str = ’Specific to Line!\n'
        for point in self.pointlist:
            str = str + `point` + '\n'
        return str
```

- Now we only inherit __init__ and addpoint from the parent
- We define a new version of __repr__ specially for line

Inspired by Richard P. Muller, California Institute of Technology presentation

Adding to parent functions

Sometimes you want to extend, rather than replace, the parent functions.

```
class line(shape):
    def __init__(self,name=”Line",intensity=0.0):
        self.intensity = intensity
        shape.__init__(self,name)
```

add additional functionality
to the constructor

call the constructor
for the parent function

Inspired by Richard P. Muller, California Institute of Technology presentation

Public and Private Data

- Currently everything in point/shape is public, thus we could do:
    - >>> p1 = point(6,0.,0.,0.)
    - >>> p1.location = ’foo'

  that would break any function that used p1.location
- We therefore need to protect the p1.position and provide accessors to this data
    - Encapsulation or Data Hiding
    - accessors are "gettors" and "settors"
- Encapsulation is particularly important when other people use your class

Inspired by Richard P. Muller, California Institute of Technology presentation

Public and Private Data, Cont.

- In Python anything with two leading underscores is private
    - __a, __my_variable
- Anything with one leading underscore is semi-private, and you should feel guilty accessing this data directly.
    - _b
  - Sometimes useful as an intermediate step to making data private

Inspired by Richard P. Muller, California Institute of Technology presentation

Encapsulated Point

```
class point:
    def __init__(self,color,x,y,z):
            self.color = color
            self.__location = (x,y,z) #position is private
    def getlocation(self):
            return self.__location
    def setlocation(self,x,y,z):
            self.__location = (x,y,z) #typecheck first!
    def translate(self,x,y,z):
            x0,y0,z0 = self.__position
            self.__position = (x0+x,y0+y,z0+z)
```

Inspired by Richard P. Muller, California Institute of Technology presentation

Why Encapsulate?

- By defining a specific interface you can keep other modules from doing anything incorrect to your data
- By limiting the functions you are going to support, you leave yourself free to change the internal data without messing up your users
  - Write to the Interface, not the the Implementation
  - Makes code more modular, since you can change large parts of your classes without affecting other parts of the program, so long as they only use your public functions

Inspired by Richard P. Muller, California Institute of Technology presentation

Classes that look like arrays

- Overload __getitem__(self,index) to make a class act like an array

```
class shape:
    def __getitem__(self,index):
        return self.pointlist[index]

>>> l1 = shape('Line') #defined as before
>>> for p in l1:        #use like a list!
        print p
>>> l1[0].translate(1.,1.,1.)
```

Inspired by Richard P. Muller, California Institute of Technology presentation

Classes that look like functions

- Overload __call__(self,arg) to make a class behave like a function

```
class gaussian:
    def __init__(self,exponent):
        self.exponent = exponent
    def __call__(self,arg):
        return math.exp(-self.exponent*arg*arg)

>>> func = gaussian(1.)
>>> func(3.)
0.0001234
```

Inspired by Richard P. Muller, California Institute of Technology presentation

More overload

- __setitem__(self,index,value)
  - Another function for making a class look like an array/dictionary
  - a[index] = value
- __add__(self,other)
  - Overload the "+" operator
  - p1= p1+ p2
- __mul__(self,number)
  - Overload the "*" operator
  - zeros = 3*[0]
- __getattr__(self,name)
  - Overload attribute calls

Inspired by Richard P. Muller, California Institute of Technology presentation

Other things to overload, cont.

- __del__(self)
  - Overload the default destructor
  - del p1
- __len__(self)
  - Overload the len() command
  - p1_l= len(p1)
- __getslice__(self,low,high)
  - Overload slicing
  - p1_s= p1[0:9]
- __cmp__(self,other):
  - On comparisons (<, ==, etc.) returns -1, 0, or 1, like C's strcmp

Inspired by Richard P. Muller, California Institute of Technology presentation

Modules

- **from** book **import** Book
- **from** book **import** *

- import math
- >>> dir(math)
- ['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']

Packages

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format
conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

`import sound.effects.echo`

from sound.effects import echo

from sound.effects.echo import echofilter

Exceptions

Exceptions

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
```

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Exception list

EnvironmentError

IOError IOError

SyntaxError IndentationError

SystemError SystemExit

ValueError

RuntimeError

NotImplementedError

Exception StopIteration SystemExit StandardError ArithmeticError OverflowError FloatingPointError ZeroDivisonError AssertionError AttributeError EOFError ImportError KeyboardInterrupt LookupError IndexError KeyError NameError UnboundLocalError

How to handle exceptions

```
try:
  # code raising an exception or not
except {NameOfError}:
  # code pick to resolve the exception or not
```

- The block ‘try’ is executed until an exception occurs
- If an exception occurs the rest of the code is skipped
- If an exception occurs not matching the exception named in the except clause, it is passed on to outer try statements
- If no handler is found, it is an unhandled exception and execution stops

try…except...else

try:

  # code raising an exception or not

except {NameOfError}:

  # code pick to resolve the exception or note

else:

  # it is useful when the code must be executed if the try clause does not raise an exception

Raising Exceptions

- `raise` statement allow the programmer to force a specified exception to occur:
  - `raise NameError('Hello')`

- You can have the `raise` statement allowing one to re-raise the exception if we don't want to handle the exception

```
try:
  # code raising an exception or not
except {NameOfError}:
  # code pick to resolve the exception or
note
```

User defined exceptions

```
class MyError(Exception):
     def __init__(self, value):
          self.value = value
     def __str__(self):
          return repr(self.value)
```

My exception occurred, value: 4

>>> raise MyError(‘Arg!’)

Traceback (most recent call last):

__main__.MyError: ‘Arg!’

Defining cleanup actions

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

Finally will be called in any circumstances
Let’s code the function divide(x,y)