Start with an Example

- Python is object oriented
- Everything is an object
- Every object has some methods
- There are no private variables/methods in python (All are public)

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

The Class Definition Syntax

```
class ClassName:
___ <statement-1>
___ .
___ .
___ .
___ <statement-N>
```

- Must be executed first to have any effect. A class definition can be inside a branch, which never even gets executed
- Usually the definitions consist of function definitions. And they have a special list of argument
- A new scope/namespace is created inside
- Once executed, an object is created

Consider the following sample class.

```
>>> class MyClass:
...     """A simple example class"""
...     i = 12345
...     def f(self):
...         return 'hello world'
...
>>>
>>>
>>> MyClass.i
12345
>>> MyClass.f
<unbound method MyClass.f>
```

Calling Class Methods

- A class is defined
- `MyClass.i` points to the variable in the class
- `MyClass.f` points to function
- But we cannot yet call that function as there is no instance of the class.
- An instance can be created by `MyClass()`

Look at the following example

```
>>> class MyClass:
...     """A simple example class"""
...     i = 12345
...     def f(self):
...         print ("hello world", self.i)
...
>>> cl = MyClass()
>>> cl.i
12345
>>> cl.f
<bound method MyClass.f of <__main__.MyClass insta
>>> cl.f()
hello world 12345
>>>
```

__init__

- Constructor of a Class
- It is called first when an instance of the class is created
- If we want to do something as the first thing, then this is the place to do it.

```
>>> class Point():
...     def __init__(self, x=0,y=0):
...         self.x = x
...         self.y = y
...
...     def __str__ (self):
...         return "".join(("(", str(self.x), ",",
                            str(self.y), ")"
))
...
>>> point1 = Point(3,4)
>>> point2 = Point()
>>>
>>> print(point1)
(3,4)
>>> print(point2)
(0,0)
```

Inheritance

- Base Class which is a common/general thing
- Derived Class which is specialised stuff
- Derived Class has all the methods of Base - INHERITANCE
- Base Class variable can keep a Derived class

```
>>> class Class1(object):
...     k = 7
...     def __init__(self, color='green'):
...         self.color = color
...
...     def Hello1(self):
...         print("Hello from Class1!")
...
...     def printColor(self):
...         print("I like the color", self.color)
...
>>> class Class2(Class1):
...     def Hello2(self):
...         print("Hello from Class2!")
...         print(self.k, "is my favorite number")
...
>>> c1 = Class1('blue')
```

```
18 >>> c2 = Class2('red')
19
20 >>> c1.Hello1()
21 Hello from Class1!
22 >>> c2.Hello1()
23 Hello from Class1!
24 >>>
25
26 >>> c2.Hello2()
27 Hello from Class2!
28 7 is my favorite number
29
30 >>> c1.printColor()
31 I like the color blue
32 >>> c2.printColor()
33 I like the color red
34 >>>
```

```
35 >>> c1 = Class1('yellow')
36 >>> c1.printColor()
37 I like the color yellow
38 >>> c2.printColor()
39 I like the color red
40 >>>
41
42 >>> if hasattr(Class1, "Hello2"):
43 ...     print(c1.Hello2())
44 ... else:
45 ...     print("Class1 has no Hello2()")
46 ...
47 Class1 does not contain method Hello2()
48
49 >>> if issubclass(Class2, Class1):
50 ...     print("YES")
51 ...
```

```
52 YES
```

Overwriting Methods

- The base class has some method
- The subclass implements the same one
- When called, based on which type, the call goes to the corresponding call
- Example below

```
>>> class FirstClass:
...     def setdata(self, value):
...         self.data = value
...     def display(self):
...         print(self.data)
...
>>> class SecondClass(FirstClass):
...     def display(self):
...         print('Current value = ', self.data)
...
>>> x=FirstClass()
>>> y=SecondClass()
>>> x.setdata("Give me the answer")
>>> y.setdata(42)
>>> x.display()
Give me the answer
>>> y.display()
```

```
18 Current value = 42
```

Abstract Classes

- Methods in the base class is not implemented.
- They must be overwritten to be able to use.
- Example below

Multiple Inheritance

Multiple inheritance is nothing but deriving from more than a single base class

```
class DerivedClass(Base1, ..., Basen):
```

The attributes/methods of base classes would be searched in a depth-first fashion, starting from the left most of the base classes.

- First look for the attribute in Base1
- Then recursively in the base classes of Base1
- Then Base2 and so on until found
- Else error

```
>>>
>>>
>>>
>>> class my_int(object):
...     def __init__(self, val):
...         self.i = val
...
...     def __repr__(self):
...         return "[" + str(self.i) + "]"
...
...     def __str__(self):
...         return "I am " + str(self.i)
...
...     def __add__(self, another):
...         return my_int(self.i + another.i)
...
...
```

```
18 ...
19 ...
20 >>> a = my_int(10)
21 >>> b = my_int(14)
22 >>>
23 >>> print(a)
24 I am 10
25 >>>
26 >>> b
27 (14)
28 >>>
29 >>> print(a+b)
30 I am 24
31 >>>
```

Other Basic Methods

`__add__`	`__iadd__`	`+` `+=`
`__div__`	`__idiv__`	`/` `/=`
`__mul__`	`__imul__`	`*` `*=`
`__sub__`	`__isub__`	`-` `-=`
`__mod__`	`__imod__`	`%` `%=`

Special Methods

Comparison Operators

`__eq__`	`==`
`__ge__`	`>=`
`__gt__`	`>`
`__le__`	`<=`
`__lt__`	`<`
`__ne__`	`!=`

Boolean Operator `__nonzero__` - could be used to enable the object ready for truth testing. LaTeX??

```
1  def movedisc(s, d):
2      d.append(s.pop())
3
4  def myprint():
5      print msrc, mdst, mtmp
6
7  def toh(src, dst, tmp, n):
8      if n == 1:
9          movedisc(src, dst)
10         return
11     toh(src, tmp, dst, n-1)
12     movedisc(src, dst)
13     toh(tmp, dst, src, n-1)
14
15 msrc = [ i*'-' for i in range(1, 4)]
16 mdst = []
17 mtmp = []
```

```
18 toh(msrc, mdst, mtmp, len(msrc))
```

Exceptions

- Exceptions are some kind of error reporting tools
- When something unexpected happens, an exception is raised
- The programmer could decide, what to do with the error
  - Could handle the exception
  - Throw/Raise the exception to the caller
- Nice things don't come for cheap.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' ob
>>> while True print('Hello world')
  File "<stdin>", line 1, in ?
    while True print('Hello world')
                    ^
SyntaxError: invalid syntax
```

Handling Them

First `try:` then `except:`

- `try` clause (stuff between the try and except) is executed.
- If no exception occurs, the except is skipped
- On exception, the rest of `try` is skipped
  - If matches the exception specified in `except`, then does the handling as in `except`
  - Else, passes to the higher level

```
>>> while True:
...     try:
...         x = int(input("A number: "))
...     except ValueError:
...         print("Oops! Try again...")
...
A number: 23
A number: \\\
Oops! Try again...
A number: 435
A number: 45%
Oops! Try again...
A number: sd
Oops! Try again...
```

```
1 for stuff in our_simple_list:
2     try:
3         f = try_to_dosomething(stuff)
4     except A_Grave_Error:
5         print ('Something Terrible With', stuff)
6     else:
7         """Continue from Try"""
8         print ("Everything fine with", stuff)
9         go_back_home()
```

When life throws lemons?

When we get exceptions.

- One way is to handle them
- Otherwise, raise them
- The present code stops executing
- And goes back to the caller

```
>>> while True:
...     try:
...         x = int(raw_input("A number: "))
...     except ValueError:a
...         print ("Oops! Try again...")
...         raise
...
A number: 12
A number: we
Oops! Try again...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: invalid literal for int() with base
>>>
```

Clean it up

Python provides with a `finally` statement, which helps to clean up if something went wrong.

- First do the try part
- Then do the `finally` part
- If exception happened, then handle the correspoding exception, then do the `finally` part.

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>>
```

```
>>>
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'st
```

Exceptions Are Classes

- Exceptions are classes too
- One can creat his/her own exceptions
- An exception can be saved in a variable for further use.
- Example below

```
>>>
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print('My exception, value:', e.value)
...
My exception, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

```
1  def arraywithproducts(A):
2      op = (1 for i in range(len(A)))
3      lp = rp = 1
4      for i in range(len(A)):
5          j = len(A) - 1 - i
6          op(i) *= lp
7          op(j) *= rp
8          lp *= A(i)
9          rp *= A(j)
10     return op
11
12 array = (1, 2, 3, 4, 5, 6)
13 print(array)
14 print(arraywithproducts(array))
```

```
1
2 (sadanand@lxmayr10 ~ pffp)python < array_products
3 (1, 2, 3, 4, 5, 6)
4 (720, 360, 240, 180, 144, 120)
5 (sadanand@lxmayr10 ~ pffp)
```

```
1  def listprod(l, lst):
2      if not lst: print("Shall not happen")
3      if len(lst) == 1:
4          return ((l), lst(0))
5      else:
6          a, alst = lst(0), lst(1:)
7          sfl     = a * l
8          blst, r = listprod(sfl, alst)
9          blst    = (l * r) + blst
10         return blst, r * a
11
12 print(listprod(1, (1, 2, 3, 4, 5)))
13
14 ((120, 60, 40, 30, 24), 120)
15
```

IO - Console

- Output : We already have used `print`
- Command line arguments: `sys.argv[i]`
- While the program is running:
  There are two methods. They both return a string which was provided by the user. (By hitting the RET)
  - `raw_input()` : returns the input string
  - `input()` : tries to execute the input string.
    DANGEROUS: Never use this to get input from users. One could compromise the system.

  NOTE: changed in Python 3.x

Read/Write Files

- There is a `file` object in python. That is used for the file operations
- One can have a handle to a file by simply using `open('filename')`
  Something similar to the
  `FILE *fp = fopen("filename", "r")` of *C*

```
>>> f = open('/tmp/workfile', 'w')
>>> print(f)
<open file '/tmp/workfile', mode 'w' at 80a0960>
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

Where are you?

- All the operations (to see soon) happens, starting from the present "position" in the file.
- Every operation changes the position of the 'cursor' in the file. (When opening a file, the seek-position is set to be 0)
- To know where we stand now, use `f.tell()`
- To move to a specific location, use `f.seek(index)`
- One has to pay attention to close the files when it is nomore needed. Otherwise, next time it could have a wrong position. `f.close()`

Reading files

- `f.read()` - Gives the text content of the file pointed to by `f`
- `f.readline()` - Gives each line by line from the file. First call gives the first line, the next call gives the next line.
- `f.readlines()` - Gives a list of the lines in the file.
- One can also directy iterate over the lines in the file.

```
>>> f = open("test.txt")
>>> f.read()
'\nThis is a test file\nThis is the second line\n
>>> f.tell()
69
>>> f.seek(0)
>>> f.readline()
'\n'
>>> f.readline()
'This is a test file\n'
>>> f.readlines()
('This is the second line\n', 'This is the final
>>> f.seek(0)
>>> for l in f:
...         print(l)
...
```

18

19

20 This **is** a test file

21

22 This **is** the second line

23

24 This **is** the final line

Writing

- When one wants to write to a file, then the `open` call has to be given specific parameters.
  - open(f, 'w'): Writeable (seek = 0)
  - open(f, 'a'): Would be appended
  - open(f, 'r+'): Readable and Writeable
  - open(f, 'r+a'): Readable and Appendable
  - open(f, 'r'): Readable
- Without a parameter, it is automatically only readable
- Using `f.mode` one can see the mode of opening.

Writing

- `f.write(string)` : Writes to `f`
- `f.writelines(col)` : Writes each member of the col (some collective object), to the file
- `f.flush()` : Writes it really to the file from the memory. Happens with `f.close()` automatically.

Pickle

- `pickle.dump(x, f)` - dumps *x* to the file
- `x = pickle.load(f)` - reads from the file to *x*

We'll see more of Pickles and Shelves soon.

Lambda functions: what are they

Mini functions.
There are times when we need to write small functions, perhaps not necessary for a reuse of anything. Then we use lambda forms.

- Only expressions can be used. No statements
- No local variables
- Only one expression

```
>>> def f(x):
...     return x*x
...
>>> print(f(7))
49
>>> g = lambda x : x*x
>>>
>>> print(g(7))
49
```

```
>>> def makeincre(n) : return lambda x: x + n
...
>>>
>>> incr2 = makeincre(2)
>>> incr9 = makeincre(9)
>>>
>>> print(incr2(10))
12
>>> print(incr9(10))
19
>>>
>>> add = lambda a, b: a+b
>>> add(10, 13)
23
```

To Note

- Variables : A comma separated list of variables. Not in parens.
- Expression : A normal python expression. The scope includes both the variables and the local scope.

Feedback on solutions?

How shall I do? A page for it?
Codenames?

Problems

- Class for chess coins
- A Rational number Class
- Flatten a List
- Create list-number pair (0,23), (1,343), ...
- Class for a Tree (Binary) (not necessarily BST)
- Knight Problems!