1. Classes
2. Exceptions
3. Using Other Code
4. Problems

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 inst
>>> 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')
>>> c2 = Class2('red')

>>> c1.Hello1()
Hello from Class1!
>>> c2.Hello1()
Hello from Class1!
>>>

>>> c2.Hello2()
Hello from Class2!
7 is my favorite number

>>> c1.printColor()
I like the color blue
>>> c2.printColor()
I like the color red
>>>
>>> c1 = Class1(’yellow’)
>>> c1.printColor()
```

```
I like the color yellow
>>> c2.printColor()
I like the color red
>>>

>>> if hasattr(Class1, "Hello2"):
...     print c1.Hello2()
... else:
...     print "Class1 has no Hello2()"
...
Class1 does not contain method Hello2()

>>> if issubclass(Class2, Class1):
...     print "YES"
...
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 = %s’ % self.data
...
>>> x=FirstClass()
>>> y=SecondClass()
>>> x.setdata("Give me the answer")
>>> y.setdata(42)
>>> x.display()
Give me the answer
>>> y.display()
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)
...
...
...
```

```
...
>>> a = my_int(10)
>>> b = my_int(14)
>>>
>>> print a
I am 10
>>>
>>> b
(14)
>>>
>>> print a+b
I am 24
>>>
```

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.

```
def arraywithproducts(A):
        op = [1 for i in range(len(A))]
        lp = rp = 1

        for i in range(len(A)):
                j = len(A) - 1 - i
                op[i] *= lp
                op[j] *= rp
                lp *= A[i]
                rp *= A[j]

        return op


array = [1, 2, 3, 4, 5, 6]
print array
print arraywithproducts(array)
```

```

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

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(raw_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...
```

```
for stuff in our_simple_list:
  try:
    f = try_to_dosomething(stuff)
  except A_Grave_Error:
    print 'Something Terrible With', stuff
  else:
    """Continue from Try"""
    print "Everything fine with", stuff
    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:
...         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 do 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 occurred, value:', e.
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

Import Statement

- No one can write all the code he/she needs.
- No need to re-invent the wheel
- Use `import` statement of Python
- Equivalent of `#include` of **C**

```
>>> import math
>>> math.pow(5, 2)
25.0
>>> math.pow(2, 5)
32.0
>>> from math import pow
>>> pow(3, 4)
81.0
```

Some nice libraries

`re`	Regular expression operations
`numbers`	Numeric abstract base classes
`math`	Mathematical functions
`cmath`	Functions for complex numbers
`decimal`	Decimal fixed & floating point math
`random`	Generate pseudo-random numbers
`os`	Miscellaneous OS interfaces
`io`	Core tools for streams
`time`	Time access and conversions
`os.path`	Common pathname manipulations

- Class for chess coins
- A Rational number Class
- Flatten a List
- Class for a Tree (Binary) (not necessarily BST)