__main__

Python has a built-in variable `__name__` when a file is run.
When a file is directly run, the value is this variable would be `__main__` when it is inside that file.
This can be sometimes very useful.

```
1
2  (sadanand@lxmayr10 * tmp)cat boo.py
3
4  def foo(fr):
5          print "Hello World from " + fr
6
7  if __name__ == "__main__":
8          foo("__main__")
9
10 foo("__outside__")
11
12 print __name__
13
14 (sadanand@lxmayr10 * tmp)python  boo.py
15 Hello World from __main__
16 Hello World from __outside__
17 __main__
18 (sadanand@lxmayr10
```

```
Python 2.6.1 (r261:67515, Jan 20 2009, 08:31:22)
(GCC 4.2.1 (SUSE Linux)) on linux2
Type "help", "copyright", "credits" or "license"
>>>
>>> import boo
Hello World from __outside__
boo
>>>
>>> boo.foo("Prompt")
Hello World from Prompt
>>>
>>>
(sadanand@lxmayr10 * tmp)
```

The main function

One can use this functionality to write "main" functions to be called from the `if` condition.

Pickle

- Module in python
- Serialisation and de-serialisation of python objects
- Serialisation : converting to a byte stream.
- The reverse to get the object back.

Pickling

- Marshalling [1]
- Serialisation
- Flattening
- Pickling / Unpickling

[1] Nothing to do with the object Marshal

cPickle and Marshal

- cPickle is the very same module implemented in C
- cPickle, yes, it is fast: about 1000 times.
- Pickle keeps track of serialisation and there is no repeated serialisation (unlike marshal)
- Shelve (for dictionaries)

How to Pickle?

- `pickle.dump(obj, file)`
- `pickle.load(file)`
- `pickle.dumps(obj)`
- `pickle.loads(str)`

A write permission to the file is required for the `dump` to work. Also, the file should have `read` and `readline` functions implemented for the `load` to be functional.

What All?

- None, True, and False
- integers, long integers, floating point numbers, complex numbers
- normal and Unicode strings
- Collections with only picklable objects
- functions defined at the top level of a module
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose __dict__ or __setstate__() is picklable

```
>>> import pickle
>>> class Foo:
...     attr = 'a class attr'
...
>>> picklestring = pickle.dumps(Foo)
>>>
>>> x = Foo()
>>> picklestring2 = pickle.dumps(x)
>>>
>>> picklestring
'c__main__\nFoo\np0\n.'
>>> picklestring2
'(i__main__\nFoo\np0\n(dp1\nb.'
>>>
>>> y = pickle.loads(picklestring2)
>>>
>>> isinstance(y, Foo)
True
```

```
>>> isinstance(x, Foo)
True
>>>
```

Threads and Processes

- Threads exist as subsets of a process (not independent)
- Multiple threads within a process share state as well as memory and other resources
- Threads share their address space
- No IPC needed.
- Context switching is typically faster

CAN SHARE GLOBAL VARIABLES

```
1  import threading
2  class MyThread ( threading.Thread ):
3      def run ( self ):
4          print 'Insert some thread stuff here.'
5          print 'It\'ll be executed...yeah....'
6          print 'There\'s not much to it.'
7
8  MyThread().start()
9  ---------------------------------------------------
10
11 Insert some thread stuff here.
12 It'll be executed...yeah....
13 There's not much to it.
```

```
1  theVar = 1
2  class MyThread2 ( threading.Thread ):
3      def run ( self ):
4          global theVar
5          print 'This is thread ' + str ( theVar )
6          print 'Hello and good bye.'
7          theVar = theVar + 1
8  for x in xrange (4):
9      MyThread2().start()
10 ---------------------------------------------------
11 This is thread 1 speaking.
12 Hello and good bye.
13 This is thread 2 speaking.
14 Hello and good bye.
15 This is thread 3 speaking.
16 Hello and good bye.
17 This is thread 4 speaking.
18 Hello and good bye.
```

Locks and Threads

- Multiple threads can communicate using a global variable
- But when two threads access the same variable at the same time?
- There are locks available

```
1  import threading
2  import time
3  from random import randint
4  class MyThread2 ( threading.Thread ):
5      lock = threading.Lock()
6      tcnt = 0
7
8      def __init__(self, gname):
9          threading.Thread.__init__(self)
10         self.name = gname
11
12     def run ( self ):
13         time.sleep(randint(1, 5))
14         print 'This is thread ' + str(self.name)
15                     + ' speaking. (call order)'
16         MyThread2.lock.acquire()
17         MyThread2.tcnt += 1
18         MyThread2.lock.release()
```

```
19             print 'Hello and good bye from thread
20                                 reached', MyThread2.tcnt
21
22 for x in xrange (4):
23     MyThread2(x).start()
```

```
1 This is thread 1 speaking. (call order)
2 Hello and good bye from thread reached 1
3 This is thread 0 speaking. (call order)
4 Hello and good bye from thread reached 2
5 This is thread 3 speaking. (call order)
6 Hello and good bye from thread reached 3
7 This is thread 2 speaking. (call order)
8 Hello and good bye from thread reached 4
```

Sockets

In python, objects can be send from sockets to sockets with the help of the Pickle Module.
The code snippet in the next slide explains this.

```
1 Client Side:
2
3 pickledStuff = pickle.dumps (PickleableObject)
4 self.channel.send (pickledStuff)
5
6
7 Server Side:
8 x = pickle.loads(client.recv(1024))
```

Speed-up Lists

- `array` : Homogenious entries. Limited space than 16 bytes for every item
- `deque` : More efficient in cases of append and left deletion/pop
- `bisect` : Keep it sorted. And do it while insertion.
- `heapq` : Maintain a heap

```
>>> from array import array
>>> a = array('H', (4000, 10, 700, 22222))
>>> sum(a)
26932
>>> a[1:3]
array('H', (10, 700))


>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1
```

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400,
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400


>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)
>>> heappush(data, -5)
>>> [heappop(data) for i in range(3)]
[-5, 0, 1]
```

Processes and Pipes

- When the client and server are running in the same system, we can use pipes.
- They can be used as files
- `os.popen(cmd, [mode, [bufsize]])` : Returns a pipe which is an `stdout` for cmd, from where the output can be read
- `os.popen2(cmd, [mode, [bufsize]])`: Similar, but an `stdin` too.

```
1  from __future__ import with_statement
2  from contextlib import closing
3  import os
4  def ls(dir):
5      with closing(os.popen("ls %s" % dir)) as pipe
6          for line in pipe:
7              yield line
8
9
10 for filename in ls("/tmp"):
11     print filename
```

Graph Isomorphism

To check whether two given graphs G and H are isomorphs, when we know the mapping f from G to H (w.l.g), All we need to do is confirm that the mapping is a bijection.
i.e, check for every node $g \in G$ that, $h = f(g) \in H$ is unique.
Also, one has to confirm that the set of edges too satisfy this property. i.e, $e_{iG} \in E_G$ has a unique $e_{iH} \in E_H$.

```
1  def isomorph(self, other, foo):
2      gnodes = self.nodes.keys()
3      hnodes = other.nodes.keys()
4      if len(nodes) != len(hnodes) : return False
5      filtered = filter(lambda v: foo(v) not in set
6      if filtered: return False
7
8      HEDGES = set((edge for edge in other.edges())
9      for (u, v) in self.edges():
10         hedge = (foo(u), foo(v))
11         if hedge not in hedges:
12             return False
13         hedges.remove(hedge)
14
15     return False if hedges else True
```

Switch Case .. or Almost the Same

- Python doesn't provide switch case
- In many cases we can still make use of python constructs to bypass `if..elif..elif..`
- The key is function pointers

```
1  def key_1_pressed():
2    print 'Key 1 Pressed'
3
4  def key_2_pressed():
5    print 'Key 2 Pressed'
6
7  def key_3_pressed():
8    print 'Key 3 Pressed'
9
10 def unknown_press():
11   print 'Unknown Key Pressed'
12
13
14 def dealkey_traditional(keycode):
15   if   keycode == 1:
16     key_1_pressed()
17   elif keycode == 2:
18     key_2_pressed()
```

```
19      elif keycode == 3:
20          key_3_pressed()
21      else:
22          unknown_key_pressed()
23
24  def dealkey_unusual(kc):
25      functions = {1: key_1_pressed,
26                   2: key_2_pressed,
27                   3: key_3_pressed}
28      functions.get(kc, unknown_press)()
29
30  dealkey_unusual(3) --- Prints Key 3 Pressed
31  dealkey_traditional(4) --- Prints Unknown Key Pre
```

Problems

- Server Client - Sockets, Threading, Sending data with Pickle Client sends some datatype, Server sends back the length of the object
- Server Client - Pipes
- Finish the search engine: Use AND OR -/MINUS operators to do the search.