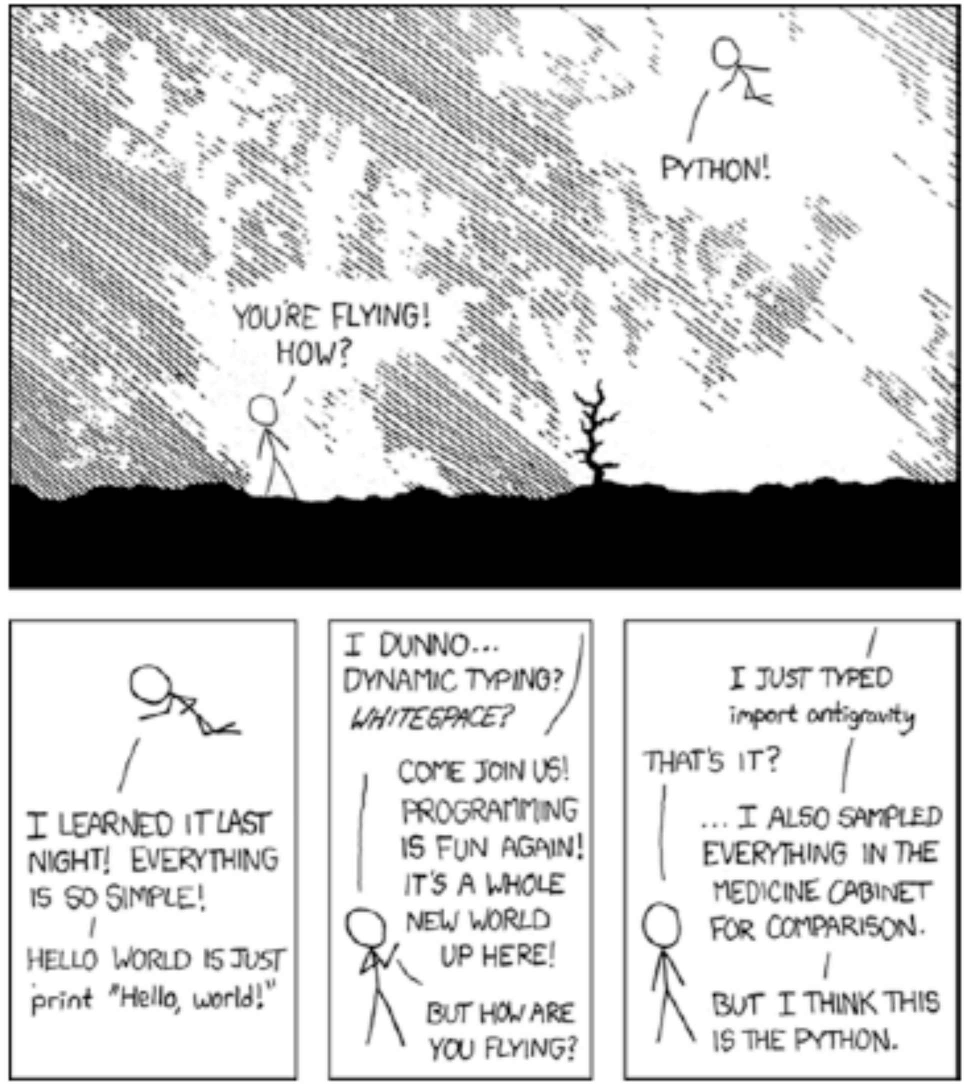
CS3101.3 Python: Lecture 4



source: http://xkcd.com/353/

Agenda

- Exceptions
- Object oriented Python
- Library demo: xml-rpc

Resources

- http://docs.python.org/tutorial/errors.html
- http://docs.python.org/tutorial/classes.html
- http://docs.python.org/library/xmlrpclib.html

Project Proposal

- One page document describing
  - Your intention, the basic idea, why it's useful or fun
  - Relevant libraries
  - Any challenges or difficulties
- Be specific
- Effort should be comparable to that you'd put into a final
- Represents most of your grade

Extra credit solution: Week 2

- Homework's 2 and 3 will be graded on course works by midday Friday
- See me if for any reason you have not submitted them
- Recurrent computer science problem:
  - Toolkit syndrome
  - Tendency to throw solutions at problems
  - Examine the instance first!
  - Reduce the problem

77				
55	34			
12	98	44		
54	11	55	43	
76	32	82	23	51

77				
55	34			
12	98	44		
54	11	55	74	
76	32	82	23	51

Exceptions

- Python's emphasis
- Use exceptions when and where they make a program **simpler**, more **robust**, more **readable**
- OK to use frequently
- OK to use when not signally an abnormal error
- Contrast with JAVA
  - Heavier weight

Raising exceptions

- An exception is an object which signifies an error or anomaly
- Terminology
- **Raising** exceptions: Python raises exceptions following the occurrence of an abnormal condition
  - Exception object is created and passed to the exception propagation mechanism
- Your code can also explicitly raise exceptions
- Equivalent terminology to **throwing** in JAVA

Handling exceptions

- Exceptions are **handled** (or caught in JAVA terminology) when code accepts an exception object from the propagation mechanism and takes some action
- In the event no code handles the exception, it proceeds all the way up the exception stack and terminates the program with an error
- Handling exceptions gives a program the ability to continue despite anomoly

Python Exceptions

- Notice that iterators signal an empty container with the StopIteration exception in response to a .next() method
- This event is neither an error or anomaly
- We'll cover Python specific error handling and checking strategies, as well as the standard logging module of the standard library

Raising (**throwing**) exceptions

Raising Exceptions

- Exceptions communicate errors and anomalies
- When problems are detected, exceptions are raised / thrown
- Your code can explicitly raise exceptions
- Exceptions are instances of the exceptions class
- The next catcher with the right sized glove catches the exceptions, or the game ends

Handling (**catching**) exceptions

Handling Exceptions

- Handling exceptions means receiving the exception object from the propagation mechanism
- If exceptions are uncaught, they are printed to std err and the program exits
- Exceptions nest!
  - You may check and catch for exceptions within other try blocks
- Exceptions propagate up namespaces / scope
  - If it's missed, the next catcher has a shot at it

Try, except, finally statements

- The try statement begins an exception-handling block
- Compound statement
  - Try followed by one or more except clauses
  - Try followed by one or more except clauses followed by an else
  - Try followed by exactly one finally clause
- Exception handlers
  - The body of each except statement handles a specific exception

Try, except, else statements

- If a try statement contains several except clauses, the exception-propagation mechanism will test each of them in order
- Handle specific cases first!
- If the final except clause lacks an expression, it will handle an exception of any type
- Beware of “bare except” statements
- Else executes only when the try clause normally terminates
- Useful to avoid accidentally handling unexpected exceptions

Syntax definition

- Try:
  - Statement(s)
- Except [expression [, target]]:
  - Statement(s)
- [else:
  - Statement(s)]

Basic exception control flow

Try / Except

```
def questionable():
    open('/')
IOError: Permission denied

def better():
    try:
        open('/')
    except IOError:
        print ‘caught a problem‘

caught a problem
```

Try / Except / Else

```
def ok():
    try:
        f = open('test', 'wb')
        print ‘success’
    except IOError:
        print 'doh'
    else:
        print 'life goes on‘

success
life goes on
```

Finally

- Try:
  - Statement(s)
- [Except…]
- Finally:
  - Statement(s)
- Finally is a clean up handler
- Specifies code which is guaranteed to run regardless of whether an exception occurs in the try block
- Useful to close database connections, files, etc

Finally

Why finally?

- Good for code that must executre regardless of whether execution completed normally
- Clean-up handler
- Occurs even if a return statement is encountered within the try and the function exits

Try / Except / Finally

```
def foo():
    try:
        f = open('test', 'wb')
        print 'opened'
    except IOError:
        print 'doh'
    finally:
        print 'closed'
        f.close()
```

Exception propagation

- When an exception is raised normal control flow is superseded by the exception propagation mechanism
- A raised exception is handled by the first try block with a matching except clause
- If an exception is raised without a try clause, or in a try clause without a matching except clause, it propagates up the call stack stack until either being caught, or terminating the program
- You can catch exceptions arbitrary deep exceptions produced by function calls

Exceptions: Themes

- Try to keep try / except cases as narrow as possible
- **In small programs: it's often better just to fail with the exception**
  - In most scripts, this is often the most efficient case
- In sensitive software: you may use broader try / except clauses, covering large segments of code
- You may also use Tkinter to nicely display errors on the screen

Exceptions: LBYL vs. EAFP

"Look before you leap"

"It's easier to ask forgiveness than permission" – Admiral Grace Hooper

Exceptions: LBYL vs. EAFP

“Look before you leap”

Why is this bad?

```
def safe_divide(a,b):
    if b == 0:
        raise ZeroDivisionError,\
            "yuck"
    return a / b


ZeroDivisionError: yuck
```

Exceptions: LBYL vs. EAFP

“Look before you leap”

Why is this bad?

- …

Exceptions: LBYL vs. EAFP

"Look before you leap"

Why is this bad?

- Checks diminish the readability of the **common case**
- Exceptions are usually rare, why perform the work up front?

Exceptions: LBYL vs. EAFP

Why is this good?

```
def save_divide(a,b):
    try:
        return a / b
    except ZeroDivisionError:
        print 'yuck'
        return
```

"It's easier to ask forgiveness than permission" – Admiral Grace Hooper

Exceptions: LBYL vs. EAFP

Why is this good?

- Readability
- Emphasizes the common case

Why is this dangerous?

- …

"It's easier to ask forgiveness than permission" – Admiral Grace Hooper

Exceptions: LBYL vs. EAFP

Why is this good?

- Readability
- Emphasizes the common case

Why is this dangerous?

- Don't cast too wide a net
- If you unintentionally catch errors raised in interior functions: you'll obfuscate them

"It's easier to ask forgiveness than permission" – Admiral Grace Hooper

Exceptions: Assert

```
def homer_date(w):
    assert (w != 'selma'), ‘omg’
    print "it's a date!"

homer_date('marge')
it's a date!

homer_date('selma')
AssertionError: omg
```

Exceptions: Assert

- The assert statement issues an AssertionError if the test fails
- A great way to document your programs – more robust than comments
- No performance hit - ignored when your code is run with the –O flag
- Self-documenting

Exceptions: Defining your own

```
class DateError(Exception):
    "Used to protect homer"

def homer_date(w):
    if w == 'selma' or w == 'patty':
        raise DateError, w
    else:
        print 'woohoo!'

homer_date('marge')
Woohoo!

homer_date('patty')
DateError: patty
```

Standard Exceptions

- Always reuse a standard exception before defining your own
- For the complete hierarchy, see
- http://docs.python.org/library/exceptions.html

Exceptions nest

Multiple except blocks

```
try:
    foo()
except Exception1:
    handle_1()
except Exception2:
    handle_2()
```

Within one another

```
try:
    foo()
    try:
        bar()
    except:
        handle_1()
    except:
        handle_2()
except:
    ..
```

Logging

- Use Python's extensive logging module, 'import logging'
- Very powerful and complex, however, you can get away with a basic subset to handle your needs
- Emit messages by 'logging.debug('my message'), or 'logging.warning('…')
- Priority hierarchy
  - DEBUG < INFO < WARNING < ERROR < CRITICAL
- You can specify behavior for each
- See: http://docs.python.org/library/logging.html
- FYI, you can also rebind sys.stderr to any file object to record uncaught exceptions which terminate your program

Object oriented Python

- Python is object oriented, but provides support for other programming paradigms
- Thus far we've been procedurally oriented
- Object oriented Python provides all the usual features – inheritance, polymorphism, etc
- Select the most appropriate style for your project

Object-Oriented Python: Themes

Overview

- Python is object oriented, but doesn't force the paradigm on the developer
- Thus far we've covered procedural programming, with some functional tools
  - Modules and functions
- Idea is to select the most appropriate paradigm for your programs

When is OO suitable?

- When you want to separate state from behavior
  - State == data
  - Behavior == code
- When you'd like to employ oo paradigms
  - Inheritance
  - Polymorphism
- When you're writing large code or code **for use by other developers**

Classes and instances

- Classes are user defined types, instantiated as objects
- Characteristics of Python classes
  - Can be called as if they were functions
  - Has a set of named attributes
  - Attributes can be data or functions (functions of classes are known as methods)
  - Classes inherit from other classes – equivalent to delegating functionality not found in a child class to the parent

Classes and instances

- Python classes as just like other objects
  - They are valid as arguments to functions, return values of methods, placed in containers, or used as attributes of other classes
- First-class objects (good concept to know)
- The class statement
- class name (base-classes):
  - statement(s)

Inheritance

- class name (base-classes):
  - statement(s)
- Base-classes are a comma seperated set of super classes from which this class inherits
- Base classes are optional, just use close parenthesis if this class doesn't extend another
- Subclass is of course transitive (e.g., if hammer subclasses tool, and tool subclasses object, then hammer is an object)
- Inheriting from 2+ classes, conflicts go to the left most inherited class

Class body

- class name (base-classes):
  - statement(s)
- The class body follows in the statements, and executes immediately upon instantiation
- Important: the class statement doesn't create the class, but only defines the specification and initial implementation

Attributes

- Attributes are specified by binding a value to an identifier within a class
- Class foo(object):
  - x = 1
- Function definitions occur similarly in the class body
  - def bar():
    - …
- Implicit attributes:
  - Begin with double underscores
  - __name__
  - __bases__
  - __dict__ (a class specific dictionary attributed used to store all other attributes)

Class private variable

- Occur via renaming
- Names beginning with a double underscore are replaced by the compiler with _classname__ident
- Decreases risk of accidental data sharing
- A convention that's up to the programmers to respect

Encapsulation

- There's no difference between a class attribute created in or outside of the class body
- With respect to encapsulation, private class variables are signified with two underscores (a bit ugly)
- Encapsulation is not Python's strong suit. From the Python doc:
- "Note that the mangling rules are designed mostly to avoid accidents; it still is possible for a determined soul to access or modify a variable that is considered private"
- See http://www.python.org/doc/2.5.2/tut/node11.html

```
class Foo():
    __x =0

print Foo.x

AttributeError: class Foo has no
    attribute 'x'
```

Instance and initialization

- To create an instance of the class, the syntax is identical to calling the class as if it were a function
- myInstance = Foo()
- Use the built-in “is-instance(I,C)” function as needed
- Calling a class object invokes the __init__ method on the new instance, deferring to the superclass if necessary
- Purpose of __init__: to bind attributes of the newly created instance, in Python you can of course bind or unbind attributes outside of __init__ as well

Class documentation

- As always, the first string in a class is the documentation string

Quick examples

Object-Oriented Python: Concepts

Who can define these?

- Classes / Instances
- Inheritance
- Composition
- Polymorphism, Overriding
- Operator overloading
- Encapsulation

Definitions

- Classes?

Object-Oriented Python: Concepts

Who can define these?

- Classes / Instances
- Inheritance
- Composition
- Polymorphism, Overriding
- Operator overloading
- Encapsulation

Definitions

- Classes: serve as instance factories. Their attributes provide behavior - data and functions – which are inherited by all instances generated from them.
- Instances?

Object-Oriented Python: Concepts

Who can define these?

- Classes / Instances
- Inheritance
- Composition
- Polymorphism, Overriding
- Operator overloading
- Encapsulation

Definitions

- Classes: serve as instance factories. Their attributes provide behavior - data and functions – which are inherited by all instances generated from them.
  - A boat.
- Instances: an instantiation of a class, representing an object in the world
  - A particular ship.
- Inheritance?

Object-Oriented Python: Concepts

Who can define these?

- Classes / Instances
- Inheritance
- Composition
- Polymorphism, Overriding
- Operator overloading
- Encapsulation

Definitions

- Inheritance: creating a new class by extending a super (parent) class results in inheriting it's behavior and attributes – multiple inheritance supported
  - "Catamaran" extends "hull", defining new behavior
- Composition?

Object-Oriented Python: Concepts

Who can define these?

- Classes / Instances
- Inheritance
- Composition
- Polymorphism, Overriding
- Operator overloading
- Encapsulation

Definitions

- Inheritance: creating a new class by extending a super (parent) class results in inheriting it's behavior and attributes – multiple inheritance supported
  - "Catamaran" extends "hull", defining new behavior
- Composition: a boat is built from a hull, a rudder, a sail – new classes often contain / benefit from collections of existing ones
  - "My little sailboat" contains "my little hull", "my little sail"
- Polymorphism?

Object-Oriented Python: Concepts

Who can define these?

- Classes / Instances
- Inheritance
- Composition
- Polymorphism, Overriding
- Operator overloading
- Encapsulation

Definitions

- Polymorphism: a subclass changes or specializes the behavior of their super class
  - Zebras swim differently than whales, but both are mammals
- Operator overloading?

Object-Oriented Python: Concepts

Who can define these?

- Classes / Instances
- Inheritance
- Composition
- Polymorphism, Overriding
- Operator overloading
- Encapsulation

Definitions

- Polymorphism: subclasses change or specialize the behavior of their super class
  - Zebras and whales are both mammals, but they swim() differently
- Operator overloading: changing the programmatic behavior of the standard operators to behave with classes
  - Homer + Donut = happy
- Encapsulation?

Object-Oriented Python: Concepts

Who can define these?

- Classes / Instances
- Inheritance
- Composition
- Polymorphism, Overriding
- Operator overloading
- Encapsulation

Definitions

- Polymorphism: subclasses change or specialize the behavior of their super class
  - Zebras and whales are both mammals, but they swim() differently
- Operator overloading: changing the programatic behavior of the standard operators to behave with classes
  - Homer + Donut = happy
- Encapsulation: exposing the minimum amount of data

Object-Oriented Python: Classes vs. Instances

- Calling a class as if it were a function results in an **instantiation**
- A Class has arbitrary attributes (variables) which you can bind and reference: these can be set on the fly
- Classes have **methods** which are attributes bound to functions
- **"Self"** is an automatically received first argument which provides a handle back to the class to be processed

```
class Sailboat():
    def setName(self, text):
        self.name = text
    def sayHello(self):
        print 'hello from', self.name

betty = Sailboat()
betty.setName('betty')
betty.sayHello()

martha = Sailboat()
martha.setName('martha')
martha.sayHello()

hello from betty
hello from martha
```

What happens here?

```
class Sailboat():
    def setName(self,
    name):
        self.name = name
    def sayHello(self):
        print 'hello from',
    self.name
```

```
Sailboat.name = 'betsy'
print Sailboat.name
Sailboat.setName('frodo')
```

What happens here?

```
class Sailboat():
    def setName(self,
    name):
        self.name = name
    def sayHello(self):
        print 'hello from',
    self.name
```

```
Sailboat.name = 'betsy'
print Sailboat.name
Sailboat.setName('frodo')

Betsy
TypeError: unbound
    method setName()
    must be called with
    Sailboat instance as first
    argument
```

Object-Oriented Python: Inheritance and constructors

- Classes can inherit from multiple other classes
  - Inherited in Python means that name lookup is extended to the super class if it can not be referenced locally
- Methods can take a special python defined name (i.e., __init__, __del__) which are implicitly evoked
- Init is a **constructor**, a method called whenever an instance of the class is created

```
class Sailboat():
    def setName(self, name):
        self.name = name
    def sayHello(self):
        print 'hello from', self.name

class Betterboat(Sailboat):
    def __init__(self, name):
        self.setName(name)

betty = Betterboat('betty')
betty.sayHello()

hello from betty
```

Inheritance and Composition

```
class Sailboat():
    def setName(self, name):
        self.name = name

class Cannon():
    def __init__(self):
        self.cannons = 0
    def addCannon(self):
        self.cannons +=1
    def fire(self):
        for i in
     range(self.cannons):
            print 'boom'
```

```
class Pirateship(Sailboat):
    def __init__(self, name):
        self.cannon = Cannon()
        self.setName(name)
    def capture(self, Sailboat):
        print self.name, 'captures', \
                Sailboat.name

x = Pirateship('betty')
x.cannon.addCannon()
y = Sailboat()
y.setName('selma')
x.cannon.fire(), x.capture(y)

boom, betty captures selma
```

Adding some behavior with isinstance

```
class Sailboat():
    def setName(self, name):
        self.name = name

class Cannon():
    def __init__(self):
        self.cannons = 0
    def addCannon(self):
        self.cannons +=1
    def fire(self):
        for i in range(self.cannons):
            print 'boom'
```

```
class Pirateship(Sailboat):
    def __init__(self, name):
        self.cannon = Cannon()
        self.setName(name)
    def capture(self, other):
        print self.name, 'shoots at',
        other.name
        self.cannon.fire()
        if  isinstance(other, Pirateship):
                print self.name, 'captures', \
                                other.name
        else:
            print self.name, 'sinks', other.name
```

x = Pirateship('betty'); x.cannon.addCannon(); y = Sailboat(); y.setName('selma');
x.capture(y); betty shoots at selma boom betty sinks selma

Class vs. Instance attributes

- Class attributes belong to the class, instance attributes to a **particular** instantiation
- Here, data is a class variable, shared among instances
- Assigning x.data creates an instance variable on x

```
class Shared():
    data = 1

x = Shared()
print x.data, Shared.data
1 1

x.data = 2
print x.data, Shared.data
2 1

Shared.data = 3
print x.data, Shared.data
2 3
```

Instance methods

- **"Self"** is an automatically received first argument which provides a handle back to the instance of the class to be processed
- Instance methods are associated with a particular instance, while class methods are associated with a class

```
Example.setName('foo')
TypeError: unbound method
    setName() must be called
    with Example instance as
    first argument
```

```
class Example():
    def setName(self, text):
        self.name = text
        print self.name

x = Example()
x.setName('instance call')
instance call

# equivalently
Example.setName(x, 'class call')
class call
```

Overriding super class methods

- Common patterns
- Inheritor: does not override a super class method, makes use of existing functionality as is
- Replacer: replaces that method with one of the same name
- Extender: calls the super's method, but adds functionality of its own
- Provider: fills in the template method of an abstract super class method

```
class Cannon():
    def fire(self):
        print 'boom!'

class TentativeCannon (Cannon):
    def fire(self):
        if raw_input('are you sure? ') ==
     'y':
            Cannon.fire(self)

x = TentativeCannon ()
x.fire()
are you sure? y
Boom!
```

Abstract class

- Abstract classes can be used to define an expected interface or behavior
  - Not implementation specific
- For instance, objects conforming to the file class must understand how to open in and out streams
  - regardless of whether that's over a network or to a local disk

```
class AbstractCannon():
    def delegate(self):
        self.fire()
    def fire(self):
        assert 0, 'fire must be defined!'

class LittleCannon(AbstractCannon):
    def fire(self):
        print 'boom'

class BrokenCannon(AbstractCannon):
        ...

x , z = LittleCannon(), BrokenCannon()
x.fire(), z.fire()
Boom, AssertionError: fire must be defined!
```

Operator overloading

When is overloading suitable?

- Operator overloading allows classes to intercept normal operations (+, - *, and, or, iteration, etc)
- Overloading moves classes closer in behavior to built in types
- Useful if you're developing a package to make the behavior more natural

Overloading "-"

```
class Donut():
   def __init__(self, n, q):
      self.name = n
      self.quantity = q
   def __sub__(self, n):
      print 'doh!', self.quantity - n,
      print "donut's left"

x = Donut('jelly', 5)
x -= 1

doh! 4 donut's left
```

Overloading iterators

- You may overload built in operators as well – such as __getitem__
  - Provides x.name[1] accessors
- Overloading __getitem__ provides iteration support as well

```
class Donut():
    def __init__(self, n):
        self.name = n
    def __getitem__(self, i):
        return self.name[i]

x = Donut('jelly_delicious')
for char in x.name:
    print char,

j e l l y _ d e l i c i o u s
```

XML-RPC

Exploring Python's Libraries: XML-RPC Requests

- RPC = Remote procedure call
- Message passing paradigm
- Protocol for exchanging XML structured information through webservices
- XML – a specification language for creating markup languages
- Very simple interaction

```
<methodCall>
    <methodName>getPrice
    </methodName>
     <params>
          <param>
                    <value>
          <string>GOOG</string>
                    </value>
           </param>
    </params>
</methodCall>
```

Exploring Python's Libraries: XML-RPC Responses

```
<methodResponse>
    <params>
        <param>
                <value>
<string>358.83 +0.79 (0.22%)  Feb 12 3:36pm ET</string>
                </value>
        </param>
    </params>
</methodResponse>
```

XML-RPC:

http://docs.python.org/library/xmlrpclib.html

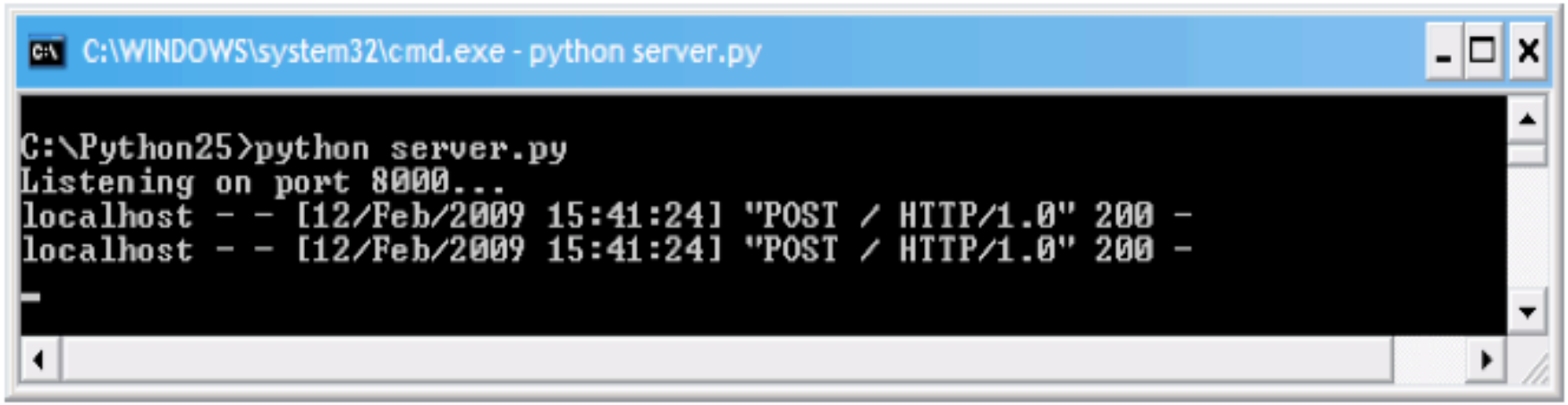
Server

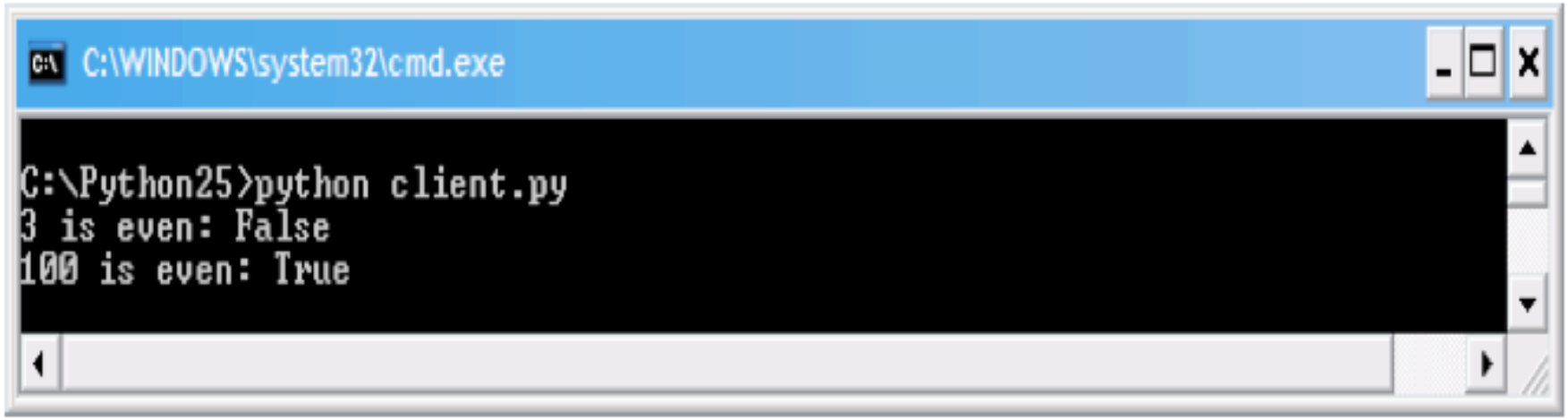
```
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer
def is_even(n): return n%2 == 0
server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000...“
server.register_function(is_even, "is_even") server.serve_forever()
```

Client

```
import xmlrpclib proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
print proxy.is_even(3)
print proxy.is_even(100)
False
True
```

XML-RPC


C:\WINDOWS\system32\cmd.exe - python server.py
C:\Python25>python server.py
Listening on port 8000...
localhost - - [12/Feb/2009 15:41:24] "POST / HTTP/1.0" 200 -
localhost - - [12/Feb/2009 15:41:24] "POST / HTTP/1.0" 200 -


C:\WINDOWS\system32\cmd.exe
C:\Python25>python client.py
3 is even: False
100 is even: True

Finding and installing libraries

http://www.goldb.org/ystockquote.html

All it takes

```
>> import ystockquote
>> ystockquote.get_price('GOOG')

357.95
```

Included Functions

- get_all(symbol)
- get_price(symbol)
- get_change(symbol)
- get_volume(symbol)
- get_avg_daily_volume(symbol)
- get_stock_exchange(symbol)
- get_market_cap(symbol)
- get_book_value(symbol)
- get_ebitda(symbol)
- get_dividend_per_share(symbol)
- ...