COMS 3101-3 Programming Languages – Python: Lecture 6

Kangkook Jee

jikk@cs.columbia.edu
Agenda

• Debugging
• Decorators
• Web development
Review

REGULAR EXPRESSION
re.match()

• Correction

```python
re.match(pattern, string, flags=0)
    Try to apply the pattern at the start of the string, returning a match object, or None if no match was found.
```

• Similar to

```python
#re.MULTILINE – regards string as single line input
re.search(^pattern, string, re.MULTILINE)
```

• To match entire line or string

```python
#search for line matches
re.search(^pattern$, string)
re.search(^pattern$, string, re.MULTILINE)
```
### Basic syntax

```
<title> this is a title </title>
```

### Pattern

```
pat = r"<title>(.*)</title>"
```

### HTML is NOT case-sensitive

```
<TITLE> this is a title </TITLE>
```

```
re.search (... , re.IGNORECASE)
```

### HTML syntax allows space elements after tag string

```
<TITLE>
> this is a title </TITLE>
```

```
pat = r"<title\s*>(.*)</title\s*>"
```
Link element

```
<A Href='HTTP://s1.s2.DOMAIN.EDU:80/~usr/script.cgi?foo=bar&answer=4'>
pat = r"<a\s*href=\s*'(.*)'" 

<A Href="HTTP://s1.s2.DOMAIN.EDU:80/~usr/script.cgi?foo=bar&answer=4">
pat = "<a\s*href=\s*'(.*)'|\"(.*)\"">

<A Href= "teaching/index.html" target="_blank">
pat = r"<a\s*href=\s*'(.*)'|\"(.*)\"">
```

• Pattern applied to real page

```
for m in re.finditer(pat, content, re.IGNORECASE):
    print m.group(1),

"#"
"http://ns1.cs.columbia.edu"
"http://www.columbia.edu"
"http://www.cs.columbia.edu/~angelos"
"projects.html"
...
DEBUGGING AND TESTING
Programs are Error Prone

- Syntax errors ← Detected by interpreter
- Errors at runtime ← Exception handling
- Incorrect programming behavior (wrong result)
  - Sometimes works, sometime not

Debugging and Testing
Debugging Tips for Scalable Project

• Utilize Python Interpreter actively
  – Program should be organized into functions / classes of reasonable sizes

• Log / Trace program behavior
  – Using print() or logging module

• assert() everywhere

• Be familiar with pdb (Python debugger)

• Establish unit testing framework
Debugging with print

- Most common way of debugging: Simply print intermediate results
- Prints all relevant information and reference to which part of the program prints
- Better: Write debugging statements to `sys.stderr`
- Comment / uncomment debugging code

```python
def mirror(lst):
    ret = []
    for i in range(len(lst)):
        ret.append(lst[-i])
        # print >> sys.stderr, \
        # "mirror: list for i={0}: ".format(i)
        # print >> sys.stderr,"{1}\n".format(lst)
    return lst + ret

x = [1,2,3]
print(mirror(x))  # Expected: [1,2,3,3,2,1]
```
logging module for debugging

- logging module to log errors and debugging messages
- Provides central control over debugging output

```python
import logging
logging.basicConfig(level = logging.DEBUG)

def mirror(lst):
    ret = []
    for i in range(len(lst)):
        ret.append(lst[-i - 1])
        logging.debug("list for i={0}: {1}" .format(i, lst[-i - 1]))
    return lst + ret

>>> mirror([1,2,3])
DEBUG:root:list for i=0: 3
DEBUG:root:list for i=1: 2
DEBUG:root:list for i=2: 1
[1, 2, 3, 3, 2, 1]
```
logging – Logging Levels

• Can output messages to on different logging levels
  – Output messages of \textit{LEVEL} and above

\begin{verbatim}
logging.basicConfig(level=logging.\texttt{LEVEL})
\end{verbatim}

\begin{tabular}{|l|l|}
\hline
\textbf{level} & \textbf{function} \\
\hline
logging.CRITICAL & logging.critical() \\
logging.ERROR & logging.error() \\
logging.WARNING & logging.warning() \\
logging.INFO & logging.info() \\
logging.DEBUG & logging.debug() \\
\hline
\end{tabular}
logging – more on logging

• Can output messages to a log file

```python
logging.basicConfig(level=logging.DEBUG,
                    filename='bugs.log')
```

• Config is valid for all modules in a program
  – Only set logfile and level once in main

• Can add and time

```python
logging.basicConfig(level=logging.DEBUG,
                    filename='bugs.log',
                    format='%(asctime)s %(message)')
```

• More on logging
  http://docs.python.org/library/logging.html
Python debugger - pdb

- Python provides a built-in debugger (module pdb)
- Allows to execute code line-by-line
- pdb allows access to program state

- Postmortem debugging

```
$ python -m pdb mirror.py
```

- Or launching pdb interactively from Python console

```
>>> from mirror import mirror
>>> import pdb
>>> mirror([1, 2, 3])
Exception!!!
...
>>> pdb.pm()
```
pdb Commands

• **b**: set breakpoint
• **n**: next line
• **r**: return from the function
• **l**: source code for current file
• **c**: continue execution until next breakpoint
Trace Generation

• Function call trace
  – What if we want to log function call / return
  – input arguments along with return values

• Recall ‘Closure’
  – Function object that contains some state defined from outside function

```python
def func_trace(func):
    # New function object wraps fun
    def new_fun(arg0):
        # Before fun is called
        print "calling with", arg0

        result = func(arg0) # Call fun

        # After fun is called
        print "return value", result

        return result

    return new_fun
```
def func_trace(func):
    # New function object wraps fun
    def new_fun(arg0):
        # Before fun is called
        print "calling with", arg0
        result = func(arg0) # Call fun
        # After fun is called
        print "return value", result
        return result
    return new_fun

>>> def inc_one(arg0):
    ...     return arg0 + 1
    ...
>>> inc_one(1)
2
>>> inc_one = func_trace(inc_one)
>>> inc_one(1)
calling with 1
return value 2
2
Trace Generation: Support arbitrary argument

- *args, **kwargs
  - to support arbitrary combinations of positional and named parameters
  - args: list containing positional parameters
  - kwargs: dictionary containing name, parameter pairs
- *, ** operators are required only when these are again become parameter to call another function

```python
def func_trace(func):

    # New function object wraps fun
    def new_fun(*args, **kwargs):

        # Before fun is called
        print "calling with", args, kwargs

        result = func(*args, **kwargs)

        # After fun is called
        print "return value", result

        return result

    return new_fun
```

```python
>>> def get_tuple(x, y):
...     return x, y
...

>>> get_tuple =
    func_trace(get_tuple)

#Positional parameter
>>> get_tuple(1,2)
calling with (1, 2) {}
return value (1, 2)
(1, 2)

#Named parameter
>>> get_tuple(x=1, y=2)
calling with () {'y': 2, 'x': 1}
return value (1, 2)
(1, 2)
```
Trace Generation: Controlling Trace Output

- Integrate trace output with Python logging infrastructure
- Redirect trace output to logging.* functions

```python
import logging

def func_trace(func):
    def new_fun(*args, **kwargs):
        # Before fun is called
        logging.Debug("calling with {0}" \
                      "{1}".format(args, kwargs))

        result = func(*args, **kwargs)

        # After fun is called
        logging.Debug("return value:" \
                      "{0}".format(result))

        return result

    return new_fun

>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> get_tuple = func_trace(get_tuple)

>>> get_tuple(1,2)
DEBUG:root:Calling with (1, 2) {}
DEBUG:root:return value (1, 2)
(1, 2)
```
Decorators

- Convenient concise way to modify classes, methods and functions
- Are a form of meta-programming (like macros in C, annotation in Java ...)
- Example uses:
  - Log all errors in a function in a special way
  - Acquire and release some resources at entry/exit point
  - Memoize previous computations performed by a function
  - Make sure only a single instance of a class exists (singleton)
  - Make a method ‘static’
  - Make a method a ‘class method’
Decorators - Syntax

• Decorators are callable objects that are applied to functions or classes

```python
@dec2
@dec1
def func(arg1, arg2, ...):
    ...
```

• is syntactic sugar for

```python
def func(arg1, arg2, ...):
    ...
func = dec2(dec1(func))
```
Decorators can take arguments

• Can call a function to get a decorator

```python
@dec(foo1, foo2)
def func(arg1, arg2, ...):
    ...
```

• is syntactic sugar for

```python
def func(arg1, arg2, ...):
    ...
func = dec(foo1, foo2)(func)
```
Example built-in decorators - `@staticmethod`

- `@staticmethod` creates a static method
- Static methods do not receive implicit ‘self’ argument
- Can be called on class and instance objects

```python
>>> class A(object):
...     @staticmethod
...     def foo(a1):
...         return a1 + 3
... 
>>> A.foo(17)
20
>>> A().foo(80)
83
```

`@classmethod` decorator performs in quite similar way, but `@classmethod` requires method to have reference to class object
Decorator: FuncTrace

- Applying FuncTrace using decorator syntax

```python
import logging
def func_trace(func):
    # New function object wraps fun
    def new_fun(*args, **kwargs):
        # Before fun is called
        logging.Debug("calling with {0}\" \\
                      "{1}".format(args, kwargs))
        result = func(*args, **kwargs)
        # After fun is called
        logging.Debug("return value:" \ 
                      "{0}". format(result))
        return result
    return new_fun

def get_tuple(x, y)
    return x, y

get_tuple = func_trace(get_tuple)
```

- Function call syntax

```python
def get_tuple (x, y)
    return x, y
get_tuple = func_trace(get_tuple)
```

- Decorator syntax

```python
@func_trace
def get_tuple (x, y)
    return x, y
```
Writing Decorators - @memoized

>>> @memoize
... def factorial(n):
...     print "#",
...     return 1 if n == 1 else n * factorial(n - 1)
... >>> factorial(5)
# # # # #
120
>>> factorial(6)
#
720

• Memoization: storing previous execution results and return cached value for the same input
# Writing Decorators - @memoized

```python
# note that this decorator ignores **kwargs
def memoize(func):
    cache = {}  # cache object

    def memoizer(*args, **kwargs):
        if args not in cache:
            cache[args] = func(*args, **kwargs)
        return cache[args]

    return memoizer
```

- `@memoize` decorator ignores named parameters (**kwargs)
- it doesn’t catch side effects – e.g., print to stdout
Writing Decorators - @singleton

• **Singleton pattern**
  – Only a single instance of a class exists
  – Instance created at the first time the class is instantiated
  – Subsequence instantiation yields reference to the same instance

```python
>>> @singleton
... class A(object):
...     pass
...
>>> x = A()
>>> y = A()
>>> x == y
True
>>> x is y
True
```
Writing Decorators - @singleton

- This is actually cheating (returns a function, not a class)
- Achieves desired behavior
- Rather complete solutions can be found from
  - [http://wiki.python.org/moin/PythonDecoratorLibrary](http://wiki.python.org/moin/PythonDecoratorLibrary)

```python
def singleton(cls):
    instances = {}

    def getinstance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return getinstance
```
WWW PROGRAMMING WITH PYTHON
A high-level view of the WWW

- WWW protocol (HTTP) implemented over TCP/IP
- IP protocol identifies a service with IP address and port numbers
- URL allow you to specify port number
  
  http://ip_address:port_num/path/to/file

<table>
<thead>
<tr>
<th>Port #</th>
<th>Default Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>ftp</td>
</tr>
<tr>
<td>22</td>
<td>ssh</td>
</tr>
<tr>
<td>23</td>
<td>telnet</td>
</tr>
<tr>
<td>80</td>
<td>http</td>
</tr>
</tbody>
</table>
HyperText Transfer Protocol (HTTP)

• Communication protocol between web client and web server
• Text based protocol
• Two main methods
  – GET: Client requests a specific resource (web-page, images ...)
    • Possibly pass some short parameter to the server
    • Most common
  – POST: Client submit data to the server and requests some result
    • Upload a file, send a message
Hypertext Markup Protocol – GET

- Client requests page by sending

```
GET /pages/test.html HTTP/1.1
```

- Server responds by providing headers and content

```
HTTP/1.1 200 OK
Date: Wed, 09 Oct 2013 17:45:34 GMT
Server: Apache
X-Powered-By: PHP/5.3.2-1ubuntu4.21
...
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/REC-html40/strict.dtd">
<html>
...
</html>
```
Hypertext Markup Protocol – GET (error response)

• Client requests page by sending

```
GET /pages/somepage.html HTTP/1.1
```

• Server responds by providing headers and content

```
HTTP /1.1 404 Not Found
Date: Wed, 29 Feb 2012 22:40:01 EST Server: Apache/2.2.9 (Debian)

Content-Type: text/html; charset=UTF-8

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/REC-html40/strict.dtd">

<html>
<body>
  <h1> Page not found on this server. </h1>
</body>
</html>
```
Low-level networking – writing servers with the sockets module

• Sockets are network endpoint associated with a specific port on a specific machine (IP address)
• Initialize a socket instance of appropriate type and bind it to address and port

```python
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', 8080))
```

• Server sockets are in listening state (must specify number of maximum requests in queue)

```python
s.listen(5)
```

• Socket blocks and waits for a connection request. Returns a new connection socket that can be read and written

```python
connection, address = s.accept()
```
Low-level networking – writing servers with the sockets module

• Once a connection is established, we can recv from and send to it

```python
conn, addr = s.accept()
while True:
    data = conn.recv(1024)
    if not data:
        break
    # Echo the data back to the client.
    conn.send(data)
socket.close()
```

• Can get a file object too

```python
f = conn.makefile('rw')
l = f.readline()  
f.write("Foo\n")  
# Need to call flush or close to send data!
f.flush()
```
import socket
import os.path

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', 8080))
s.listen(1) # Set socket to 'listening' state

while True:
    conn, addr = s.accept()
    cfile = conn.makefile('rw')
    l = cfile.readline()

    print "RECV:", l

    if l.startswith("GET "):
        pathname = os.path.join("jikk_web/", l.split()[1][1:])

    try:
        cfile.write('HTTP/1.0 200 OK\n\n')
        cfile.write(open(pathname, 'r').read())
    except IOError:
        cfile.write('HTTP/1.0 404 Not Found\n\n')
        cfile.write('Not found!')
    finally:
        cfile.close()
        conn.close()
BaseHTTPServer

• Standard library support for static webserver
• Merged to http.server in Python 3
  – 2to3 tool will automatically fix it

```python
#running webserver from command line
jikk$ python -m BaseHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

• To implement protocol you need to extend BaseHTTPRequestHandler class
Web Development

Develop user applications that run on the web

• Client-based code that runs on the client
  – Java, JavaScript, Flash, Silverlight ...
  – Python compiled to JavaScript

• Server-based code
  – Static web page: HTML
  – Dynamic web page generation
    • Common Gateway Interface
  – Web frameworks
    • For Python: Django, Zope, Pylons, TurboGears, web2py, Flask
Common Gateway Interface (CGI)

• CGI: method for web servers to delegate generation of web pages to a program
  – Server sets up environment for the script and runs script
  – HTTP POST method can be used to provide stdin for script
  – HTTP GET can pass query string to the script (from HTML forms)
  – Server returns program stdout to the client

• Normally CGI requires some web-server configuration
Python CGI scripts

• Fort test purpose, we can use a standard library CGI capable web server
  – CGI script (often) live in cgi-bin/ subdirectory

```bash
$ python -m CGIHTTPServer
cgi-bin/cgi0.py

#!/usr/bin/python

response = ""
<html> <body>
<h1> Hello World </h1> </body>
</html>"

print response
```

• Now we can browse to
  http://localhost:8000/cgi-bin/cgi0.py
HTML Forms and Query Strings

```html
<form name="query" action="query.py" method="GET">
Query Input: <input type="text" name="Input" /><br/>
Query Type:
<input type="radio" name="input" VALUE="z"/>zipcode
<input type="radio" name="input" VALUE="c"/>city <br/>
<input type="submit" value="Submit"/>
</form>
```

- When ‘submit’ is pressed, the browser uses the http GET method to send request
  
  `query.py?Input=10027&input=z`
- `attribute/value pairs` attached to script name (after ? separated by &)

Query Input: 10027
Query Type: ☑️ zipcode ☐️ city
Submit
cgi module

• Needed to access passed attribute/value pairs in the CGI script

```python
#!/usr/bin/python
import cgi
form = cgi.FieldStorage()

def make_msg(qtype):
    if qtype == "z":
        return "Input is a zipcode"
    elif qtype == "c":
        return "Input is a city name"
    else:
        return "Invalid"

qtype = form.getvalue("qtype")
msg = make_msg(qtype)

print('')
<html> <body> {0}:  {1}<br>
<a href="../query.html"> back </a> </body > </html >
'''.format(form.getvalue("input"), msg))
```
cgi module (2)

- Also support attribute/value pairs passed by POST method
- Message will not be visible from URL, it is delivered as a separate message
- `cgi.FieldStorage` takes care of reading and processing data

```html
<html ><body >
<form action="cgi-bin/comment.py" method="post">
comment:<br>
<textarea name="comment" cols="40" rows="4">Text here</textarea>
<br>
<input type="submit" value="Submit"/>
</form>
</body></html>
```

```python
#!/usr/bin/python
import cgi
form = cgi.FieldStorage()
print('''Content-Type: text/html
<html> <body> Comment: {0}</body></html>'''.format(form.getvalue("comment")))
```
cgitb module

• If a CGI script contains errors, it may crash
• Possibly prints exceptions to a server log file
• No output sent back to the client
• cgitb module sends pretty debugging output back to client

```python
import cgitb
cgitb.enable()
```

• Nice for debugging purposes! Don’t use on public websites!
Drawbacks of CGI

• Need to run a new process every time to generate a single web page
  – Think about what happens with thousands of user requests!

• Hard to represent state in a web application
  – Need to save information and re-read it

• Some security issues unless webserver is well configured and scripts are written with care
Credits

• Many thanks to Daniel Bauer
  – For his wonderful work on class materials!

• I referred a lot from his material
  – For this classes’ material and homeworks