COMS 3101-3 Programming Languages – Python: Lecture 4

Kangkook Jee
jikk@cs.columbia.edu
Review

• Advanced functions
  – Map / reduce / filter
• Class
  – Custom type definition
  – Big thing
• Modules and Packages
Assignment 3

- Part3 typo -- fixed
- Part4
  - Advanced sorting (will cover today)
  - Traveling Salesman problem (TSP)
Agenda

• Class Re-cap
• Exception
• Sorting
• Standard Library
  – os
  – sys
  – collection
  – pickle
  – urllib2
CLASS REVIEW
Class Review

• Support user-defined/custom type
• Blueprint that describes how to build objects
  – Class is an object too
• **Objects**: instantiated grouping of states(attributes) and behaviors(methods)
• **Method**: functions associated to the object and can access and manipulate the object’s states
• **Attributes**: data fields of the object for state maintenance
Custom Integer Definition

```python
class Number(object):
    def __init__(self, value=0):
        self.value = value

    def __repr__(self):
        return str(self.value)

    def __str__(self):
        return str(self.value)
```

- **Class Number**
  - constructor with single parameter (default = 0)
  - Two special methods overridden
Custom Integer Definition

```python
class Number(object):
    def __init__(self, value=0):
        self.value = value
    def __repr__(self):
        return str(self.value)
    def __str__(self):
        return str(self.value)

class MyInt(Number):
    def __add__(self, other):
        return MyInt(self.value + other.value)
    def __div__(self, other):
        return MyInt(self.value / other.value)
    def __eq__(self, other):
        return self.value == other.value
```

- Single inheritance
  - arithmetic and comparator methods are overridden
Custom Integer Definition

```python
class Number(object):
    def __init__(self, value=0):
        self.value = value

    def __repr__(self):
        return str(self.value)

    def __str__(self):
        return str(self.value)

class MyInt(Number):
    def __add__(self, other):
        return MyInt(self.value + other.value)

    def __div__(self, other):
        return MyInt(self.value / other.value)

    def __eq__(self, other):
        return self.value == other.value

class Conversion(object):
    def __init__(self, value=0):
        self.value = value

    def to_binary(self):
        return bin(self.value)

    def to_hex(self):
        return hex(self.value)
```

- **Multiple inheritance**
  - Duplicate constructors: `__init__()`
  - Method resolution order (MRO) applied
EXCEPTIONS
What happens?
if you do something stupid

Example 1: KeyError

```python
>>> city2addr={}
>>> city2addr['New York'].append(addr)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'New York'
```

Example 2: IndexError

```python
>>> lst = range(10)
>>> lst[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

• Python complains about runtime errors
  – By raising errors (Exceptions)

• Exceptions are objects
  – Built-in exceptions
  – Custom exceptions
Two Error Handling Approaches

- Check before Operation

```python
def reverse_seq0(seq):
    if (len(seq)):
        print(seq.pop()),
        reverse_seq0(seq)
    else:
        return
```

- Operation first, then handle errors

```python
def reverse_seq1(seq):
    try:
        print(seq.pop()),
        reverse_seq1(seq)
    except IndexError:
        return
```

- Reverse sequence data type
  - Need to check for empty sequence
Programs are Error Prone

• Syntax errors ← Detected by interpreter
• Incorrect programming behavior (wrong result) ← Testing (later)
  ```python
def hypotenuse(x, y): return x**2 - y**2
  ```
• Errors at runtime ← Exception handling
  – NameError: referring undefined variables
  – TypeError: operations not supported by type (class)
  – NumericError: division by zero
  – IOError: file not found, cannot write to file ...
  – ...
Exceptions

• Exception = “Message” object that indicates an error or anomalous condition
• When an error is detected, Python raises an exception
• Exception is propagated through the call hierarchy
• If the exception is not handled and arrives at the top-level:
  – Program terminates
  – Error message and traceback report is printed
Example Error and Traceback

```python
>>> getline("... ,New York, ... ")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "line.py", line 5, in getline
    insert_city(entries[7], entries)
  File "line.py", line 8, in insert_city
    citydict[city].append(entries)
KeyError: 'New York'
```

- Traceback contains the path took through the call hierarchy
- Includes module name, function name and line numbers

```python
1 citydict={}
2
3 def getline(input):
4   entries = input.split(",")
5   city entries[7]
6   insert_city(city, entries)
7
8 def insert_city(city, entries):
9   citydict[city].append(entries)
```
try ... except (1)

• If an error occurs in the block indented below try
  – Execution is interrupted at the point of error
  – Optional except block is executed if exception has the right type
  – Execution is resumed after try ... except block

```python
1 citydict={}
2
3 def getline(input):
4     entries = input.split()
5     city = entries[7]
6     try:
7         insert_city(city, entries)
8     except KeyError, ke:
9         print ('caught error', ke)
10     citydict[city] = [entries]
11
12 def insert_city(city, entries):
13     citydict[city].append(entries)

>>> getline(line)
('caught error', KeyError('New York',))
>>> citydict
{'New York': [['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']]}```

```
try ... except (2)

```python
try:
    x = bar(a)
except TypeError:  # Binding the exception
    # object is optional
    print('caught Type error.')
except ZeroDivisionError, ex:
    print('caught div0 error.')
```

- can use multiple except block for different types

```python
try:
    x = bar(a)
except (TypeError, ZeroDivisionError):
    print('caught either a type or a div0 error.')
```

- can use tuple of exception types

```python
try:
    x = bar(a)
except:
    print('caught every exception.')
```

- catch all exceptions (use sparingly)
try ... except ... else

- Optional else block is run only if try block terminates normally
  - When none of except blocks are visited
- Avoids unintentionally handling exceptions that occur in the else block

```python
try:
    x = bar(a)
    y = 72 / x
except ZeroDivisionError:
    print('caught a div0 error from bar.')
```

```python
try:
    x = bar(a)
except ZeroDivisionError:
    print('caught a div0 error from bar.')
else:
    try:
        y = 72 / x  # can cause a different div0 error!
    except ZeroDivisionError:
        print('caught another div0 error.')
```

Does not distinguish which line causes the error

Handled from different try ... except block
try ... except ... finally

- finally block is executed no matter what!
  - When the try block terminates normally
  - When an exception is caught
  - Even if break, return, continue is called or another exception is raised
    - Right place to have clean-up statements

```python
def foo(x):
    try:
        y = x[0]
        return y
    except IndexError:
        return 0
    finally:
        print("Done.")
```

```bash
>>> foo([])
Done.
0
>>> foo([42])
Done.
42
>>> foo(42)
Done.
...
TypeError: 'int' object is unsubscribe
```
Raising Exceptions

• Exceptions can be raised if internal errors occur
• Exceptions can be initiated explicitly with `raise`
Passing on Exceptions

- Can pass on Exceptions through the call hierarchy after partially handling them

```python
def foo(x):
    try:
        y = x[0]
        return y
    except IndexError:
        print("Foo: index 0 did not exist.")
        print("Let someone else deal with it.")
        raise  # Re-raise exception
```
Built-in and Custom Exceptions

- List of built-in exceptions: [http://docs.python.org/library/exceptions.html](http://docs.python.org/library/exceptions.html)
- Can write/define your own exceptions:
  - Exceptions are classes
  - Subclass any of the defined Exceptions (try to be as specific as possible)

```python
class EmptyListException(IndexException):
    """ An Exception that indicates that we found an empty list. """

def foo(x):
    try:
        y = x[0]
        return y
    except EmptyListException, ex:
        sys.stderr.write("Argument list cannot be empty.\n")
        return None
```
Using Exceptions Properly

• Write exception handlers only if you know how to handle the exception
  – i.e., it’s easy to back-off or the exception is normal behavior
• Except specific exception classes
  – rather than general ones such as Exception or StandardError (can mask unexpected errors)
• Raise informative exceptions rather than just terminating the program
• Can use exception for control flow (Recall break, continue)
  – From some other languages, this is regarded as a bad practice (Java)

Easier to Ask for Forgiveness than for Permission (EAFP)

```python
def reverse_seq1(seq):
    try:
        print seq.pop(),
        reverse_seq1(seq)
    except IndexError:
        return
```
SORTING

Examples are courtesy of Google python tutorial
sorting for list

- list supports `sort()` method implements stable sort IN-PLACE

```python
L.sort(cmp=None, key=None, reverse=False)
```

```python
>>> a = [5, 1, 4, 3]
>>> a.sort()
>>> a
[1, 3, 4, 5]
>>> a.sort(reverse=True)
>>> a
[5, 4, 3, 1]
>>> help(a.sort)
```

- You can implement custom sorting by providing comparison function to `key` or `cmp` parameter
  - `cmp` parameter is deprecated
Sorting with `sorted()`

- `sorted()`: takes a sequence data type and returns a new sequence with those elements in sorted order

```python
sorted(iterable, cmp=None, key=None, reverse=False)
```

```python
>>> a = [5, 1, 4, 3]
>>> b = sorted(a)
>>> a
[5, 1, 4, 3]
>>> b
[1, 3, 4, 5]
```
Custom Sorting with key=

- Specify key function for more sophisticated sorting
  - key function take one input return one output
  ex) built-in function len()

```python
>>> len
<built-in function len>

>>> strs = ['cac', 'aaaa', 'daa', 'bb']
>>> print sorted(strs, key=len)
['bb', 'cac', 'daa', 'aaaa']
```
Custom Sorting with **key=**

- **count_a(x)** counts the occurrence of ‘a’ from strings
  - You can generalize functions

```python
def count_a(x):
    return x.count('a')
```

```python
>>> strs = ['cac', 'aaaa', 'daa', 'bb']
>>> print sorted(strs, key=count_a)
['bb', 'cac', 'daa', 'aaaa']
```
Quiz

- Dictionary data structure
  - grading {name: grade}
  - sort keys by its values

```python
>>> grading = {'Kangkook': 59, 'Ethan': 80, 'John': 77, 
              'George': 89, 'Kontaxis': 75}
```  

- Defining a key function

```python
>>> def get_value(key):
...     return grading[key]

>>> sorted(grading, key=get_value)
['Kangkook', 'Kontaxis', 'John', 'Ethan', 'George']
```  

- Using `dict.get()` method

```python
>>> sorted(grading, key=grading.get)
['Kangkook', 'Kontaxis', 'John', 'Ethan', 'George']
```
STANDARD LIBRARY
Standard Library

• So far: structure of the programming language itself
• Python comes with a ‘battery included’ philosophy
  – A lot of built-in functionalities
  – Large standard library modules
• Will cover some import / representative modules
• See docs for more
  http://docs.python.org/library/index.html
Some Important Modules (1)

• General Purpose:
  – sys: Access runtime environment
  – collections: More container data-types
  – itertools: fancy iterators
  – time: Time access and conversions
  – math: Mathematical functions
  – subprocess: Spawn child process

• Strings
  – re: Regular expressions

• File I/O
  – os: interact with the operating system
  – os.path: pathname operation / browse FS
  – csv: read/write comma separated value(CSV) file
  – shutil: High level file operation

• GUI
  – TKinter: built-in GUI
Some Important Modules (1)

• Internet / Networking
  – urllib2: Open / access resource by URL
  – smtplib: email processing
  – SimpleHTTPServer: simple http request handler
  – xmlrpclib: XML-RPC client

• Debugging / Profiling
  – logger: built-in logging
  – pdb: Python debugger
  – trace: Trace statement execution

• Development
  – Pydoc: Document generator and online help system
  – unittest: Python unit testing framework
sys

System (i.e. Python interpreter)-specific parameter and functions
sys Module: IO Stream File Objects

- `sys.stdin`: terminal input
- `sys.stdout`: terminal output
- `sys.stderr`: error stream
  - By default stderr is printed to terminal as well
  - In UNIX/Linux/Mac: can ‘redirect’ different streams to files

**sys_inout.py**

```python
1 import sys
2 if __name__=='__main__':
3     input = sys.stdin.read()
4     sys.stdout.write("stdout: " + input)
5     sys.stderr.write("stderr: " + input)
```

```
jikk$ python sys_inout.py
abcd
stdout: abcd
stderr: abcd
jikk$ python sys_inout.py > /tmp/stdout 2> /tmp/stderr
abcd
```
sys Module: path

• `sys.path`: a list of directory locations that determines where Python searches for modules
  – Environment variable `PYTHONPATH` is appended to default path

```
jikk$ export PYTHONPATH="$PYTHONPATH:/Users/jikk/project/"
jikk$ python
Python 2.6.8 (unknown, Jul 31 2012, 14:17:35)
[GCC 4.2.1 Compatible Apple Clang 4.0 ((tags/Apple/clang-421.0.57))] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/Library/Python/2.7/site-packages', '/Users/jikk/project/']
```
sys Module: Command Line Arguments

- `sys.argv` is a list of containing command line arguments
  - `sys.argv[0]` is the name of the script
  - all other elements are arguments passed to the script
  - arguments are passed as `string`

```python
import sys
if __name__ == '__main__':
    print(sys.argv)
```

```
jikk$ python test_args.py arg0 arg1
['test_args.py', 'arg0', 'arg1']
```
OS
python interface to OS operations
File Operation with ‘os’

• ‘os’ module defines interfaces that enable interactions with operating systems
  – Most frequently used component of standard library
  – Implements majority subset of OS system call API

```python
>>> import os
>>> os.system('date')  # OS specific command
Wed Sep 9 22:16:59 EDT 2013
0
```

• ‘os.path’ sub-module defines interfaces for filename manipulation

```python
>>> os.path.isdir("/tmp")  # some folder
True
```
os.path Module – manipulate pathnames

- `os.path.abspath(path)`: Returns the absolute pathname for a relative path

  ```
  >>> os.path.abspath('python')
  '/opt/local/bin/python'
  ```

- `os.path.basename(path)`: Returns the absolute pathname for a relative path

  ```
  >>> os.path.basename('/opt/local/bin/python')
  'python'
  ```

- `os.path.getsize(path)`: Returns the size of path in byte

  ```
  >>> os.path.getsize("python")
  13404
  ```

- `os.path.isfile(path)`: Returns True if the path points to a file

- `os.path.isdir(path)`: Returns True if the path points to a directory
os Module – list, walk content of a directory

- `os.listdir(path)` lists files in a directory

```python
>>> os.listdir("/tmp")
['.font-unix', '.ICE-unix', ... , android-jikk']
```

- `os.walk(path)` returns generator object traverse sub-directories in depth-first fashion

```python
>>> w = os.walk('/tmp')
>>> loc = w.next()
>>> while w:
...     print loc
...     loc = w.next()
```
collections
High-Performance Container Datatypes
collections.defaultdict

- A dictionary class that automatically supplies default values for missing keys
- Is initialized with a factory object, that create
  - can be a function or a class object
  - can be a basic type (list, set, dict, int initializes to default value )

Counter using `dict`

```python
1 def count_seq(seq):
2     seq_dict = {}
3     for ent in seq:
4         if ent in seq_dict:
5             seq_dict[ent] += 1
6         else:
7             seq_dict[ent] = 1
8     return seq_dict

>>> count_chr('sdfs')
{'s': 2, 'd': 1, 'f': 1}
```

Counter using `defaultdict`

```python
10 from collections import defaultdict
11
12 def count_seq0(seq):
13     seq_dict = defaultdict(int)
14     for ent in seq:
15         seq_dict[ent] += 1
16     return seq_dict

>>> count_chr('sdfs')
defaultdict(<type 'int'>, {'s': 2, 'd': 1, 'f': 1})
```
collections.Counter

- Easy interface to count hashable (immutable) objects in collections (often strings)
- Once created, they are dictionaries mapping each object to its count
- Support method `most_common(n)`
- Can be updated with other counters or dictionaries

```python
>>> from collections import Counter
>>> c = Counter('banana')
>>> c
Counter({'a': 3, 'n': 2, 'b': 1})
>>> c.most_common(2)
[('a', 3), ('n', 2)]
>>> c.update({'b':1})
>>> c
Counter({'a': 3, 'b': 2, 'n': 2})
>>> c['b']
2
```
pickle
Object serialization / Data persistence
Pickle: Object Serialization

• Provide a convenient way to store Python objects in file and reload them
• Allows saving/reloading program data or transferring them over a network
• Can pickle almost everything
  – All standard data types
  – User defined functions, classes and instances
  – Works on complete object hierarchies
  – Classes need to be defined when un-pickling

```python
import pickle
f = open('zip2addr.pickle','w')
pickle.dump(zip2addr, f)
f.close()

f = open('zip2addr.pickle','r')
zip2addr = pickle.load(f)
f.close()
```
Pickle: Protocols and cPickle

- Normally pickle uses a plaintext ASCII protocol
- Newer protocols available
  0: ASCII protocol
  1: old binary format (backward compatible)
  2: new protocol (≥ Python 2.3, more efficient)
- More efficient reimplementation of Pickle in native C
  – Always use this for large object hierarchies (up to 1000x faster)

```python
import cPickle as pickle
f = open('zip2addr.pickle','wb')
zip2addr = pickle.dump(zip2addr, f, protocol=2)
f.close()
```
urllib2
Open / Access resource by URL
urllib2: Fetch URLs

- URL: uniform resource locator
  - Support various Internet protocols: http, ftp, file ...
- urllib2 enables
  - fetch internet resources located by URL
  - Interface to modify request headers

#URL for file
file://localhost/Users/jikk/jikk_web/index.html

#URL for HTTP

#URL for ftp
ftp://user:password@host:port/path
urllib2: Getting Contents

- urllib2.Request()
  - returns URL request object that you are making
  - Can specify extra information (meta) associated to the request
    ex) browser type, cookie ...
- urllib2.urlopen()
  - Open connection to the host and return response object

# basic usage
import urllib2
req = urllib2.Request("http://www.columbia.edu")
content = urllib2.urlopen(req)
lines = content.readlines()
for line in lines:
    print line
urllib2: Download files

• URL response operates as file object
  – Can write its contents to another file object
• shutil module provides easier interfaces to manipulate files

```python
#downloading large binary file
output = open("output.zip", "wb")

CHUNK_SIZE=1024
buf = req.read(CHUNK_SIZE)
len(buf)
while len(buf):
    output.write(buf)
    buf = req.read(CHUNK_SIZE)

#copying file using shell utilities(shutil)
import shutil
output = open("output1.zip", "wb")
shutil.copyfileobj(req, output)
```