Week 8

Unit project Part 1

Agenda

- Pickle
- The chat system: an overview
- UP1 tasks
- Terminal/CMD commands

Pickles & Object Serialization

Pickling

- We can directly work with binary files with the pickle module
- To use: import the pickle module, use the appropriate file mode
  - ‘r’ -> ‘rb’ (read binary)
  - ‘w’ -> ‘wb’
- pickle.**dump**()
  - Writes an object directly to file as its native data type
    - (***not*** a string!)
- pickle.**load**()
  - Reads the first available binary object in the file and returns it

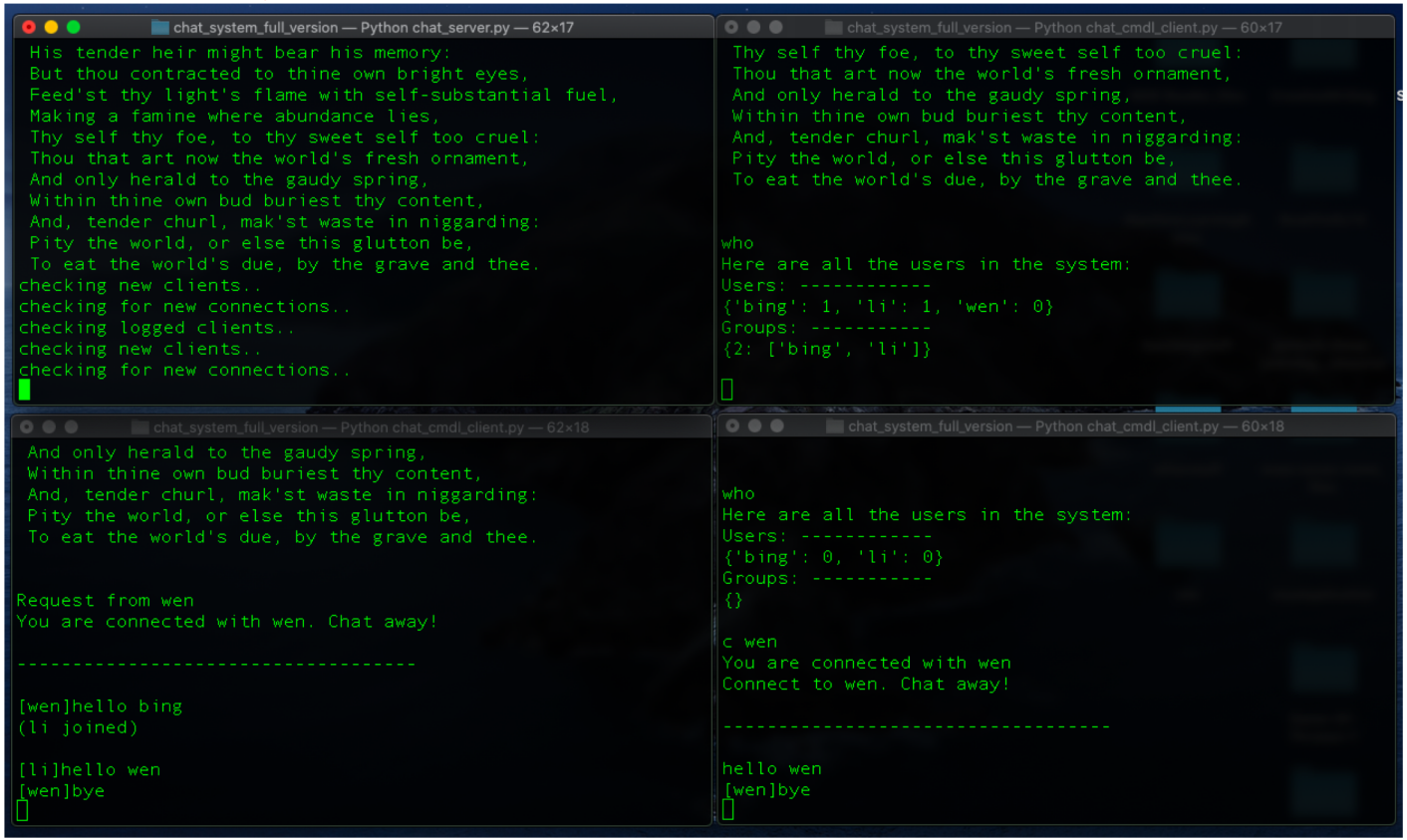
Pickling Analysis

```
# singer : album
f = open("albums_ver1.txt", "w")
f.write("Adele : 25, Bieber : 2U, Swift : 1989")
f.close()

input_f = open("albums_ver1.txt", 'r')
albums = input_f.read()

print(albums)
print(type(albums))

>>> Adele : 25, Bieber : 2U, Swift : 1989
>>> <class 'str'>
```

```
# pickle examples
import pickle

albums = {"Adele":"25", "Bieber":"2U", "Swift":"1989" }
output_file = open("albums_ver2.txt", "wb")
pickle.dump(albums, output_file)
output_file.close()

input_file = open("albums_ver2.txt", 'rb')
albums = pickle.load(input_file)

print(albums)
print(type(albums))
>>> {'Adele': '25', 'Bieber': '2U', 'Swift': '1989'}
>>> <class 'dict'>
```

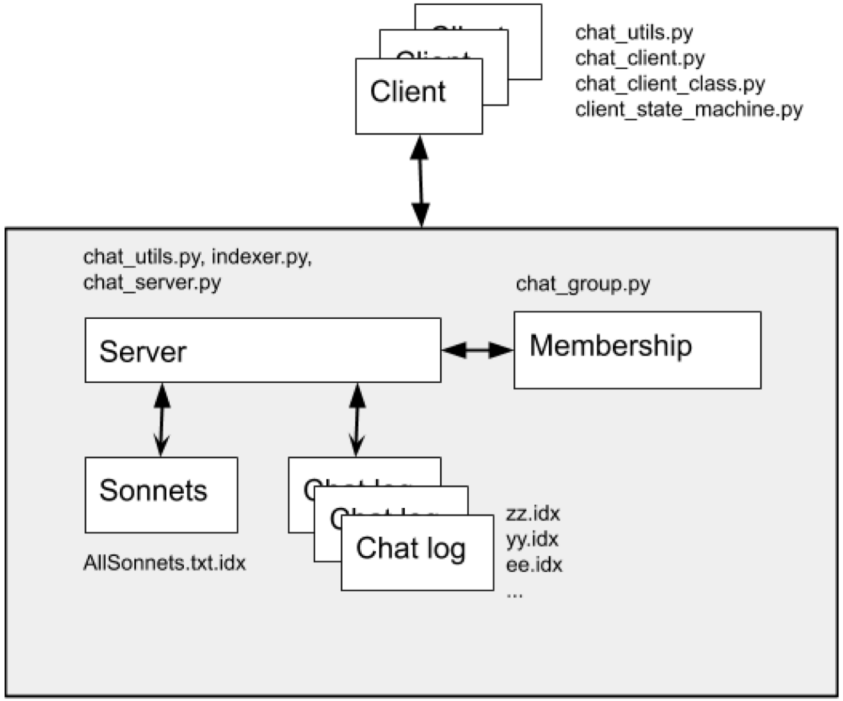
We get a **dict** type object, not a **string** type

The chat system: a demo



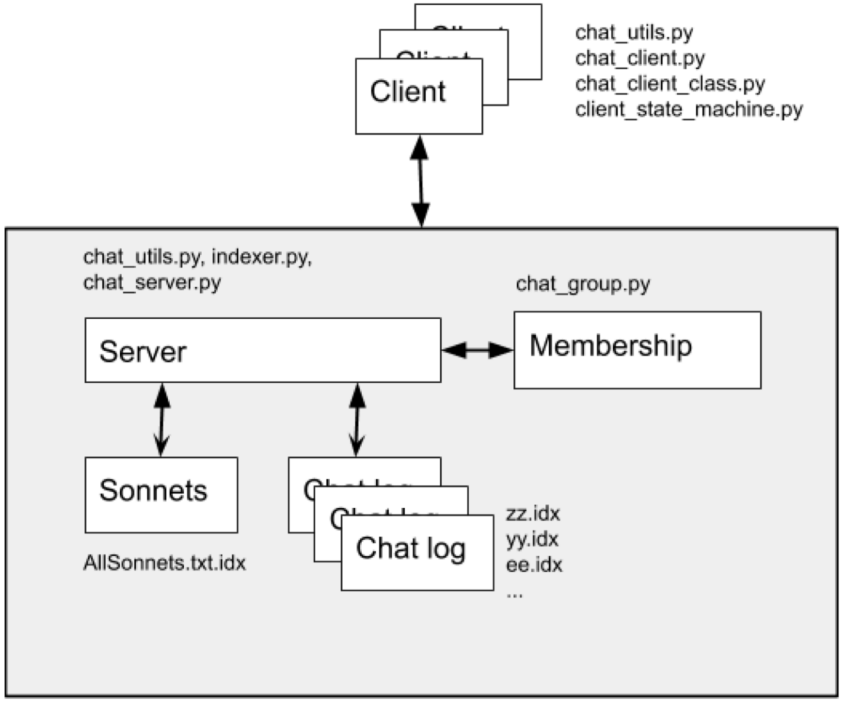
The architecture

- A typical distributed client-server system
  - Clients interact with a central server
  - WeChat is built on this too.
- The server passes messages back and forth.
  - Clients interact with each other as if directly
  - It also adds some other functionalities such as indexing history.



The architecture

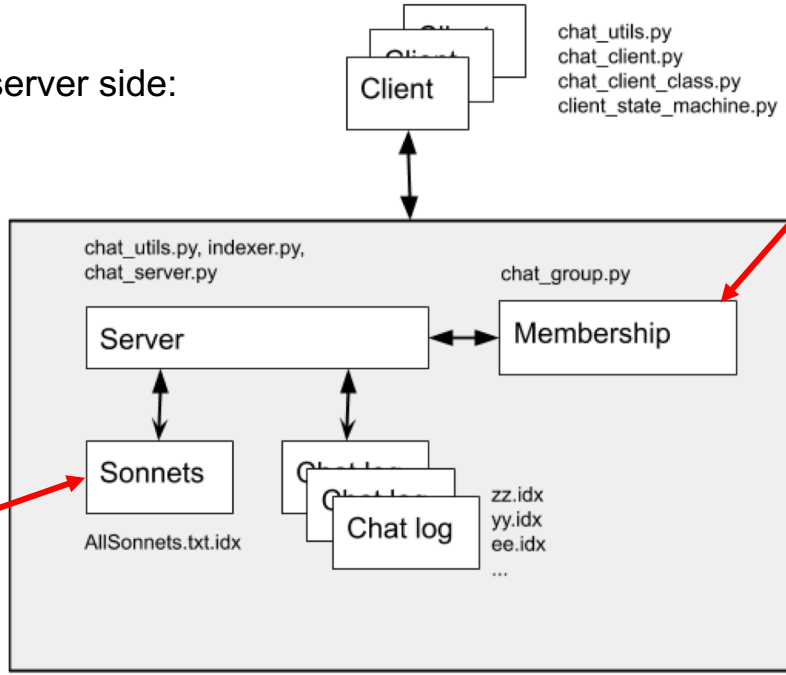
- There are multiple clients, each of them is either idle, or actively participates in one chat session with a group of other clients.
    - Compared with WeChat, our system is simple: It allows chatting in one group only.



Files of the server side

Files that make up the system server side:

- chat_server.py
- indexer.py



A membership management module

- e.g., to look at who is chatting with whom
- chat_group.py

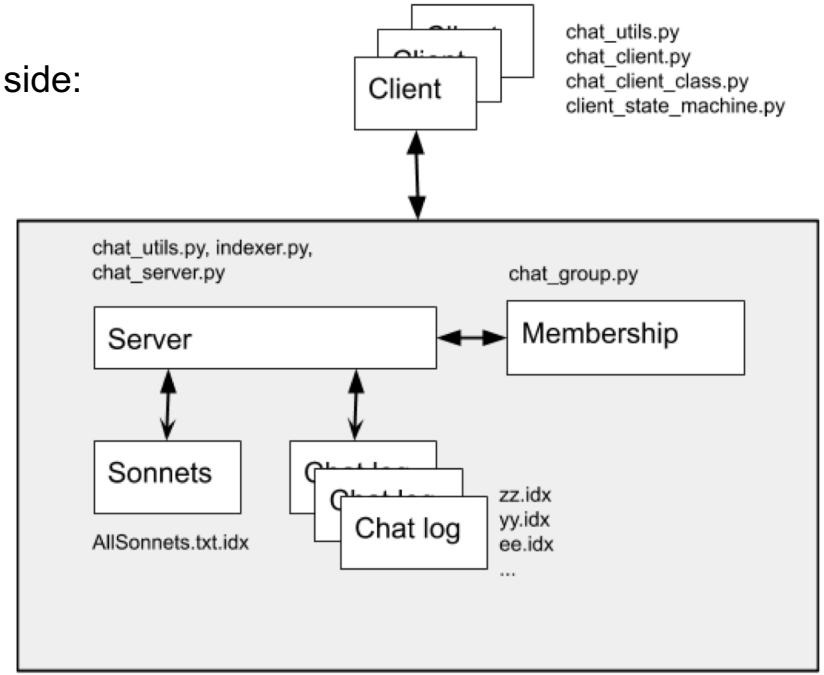
A sonnet database

- A user can ask for a poem when she is not chatting.
- AllSonnets.txt.pk

Files of the client side

Files that make up the system client side:

- chat_client.py
- chat_client_class.py
- client_state_machine.py



UP1: the files

- indexer_student.py: you need to modify this file
- AllSonnets.txt: a small collection of Shakespeare's sonnets
- roman.txt: provide the integers to roman numerals conversions
- roman2num.py: A Python module for converting integers to roman numbers
- roman.txt.pk: A binary file which contains a roman number-integer mappings
- p1.txt: the first sonnet, just for testing purposes.

You will implement functions for two classes:

- the base class `Index`
- the derived class `PIndex`

Please read the specifications of UP1 for details.

Index class

```
class Index:
    def __init__(self, name):
        self.name = name
        self.msgs = [];
        self.index = {}
        self.total_msgs = 0
        self.total_words = 0

    def get_total_words(self):
        return self.total_words

    def get_msg_size(self):
        return self.total_msgs

    def get_msg(self, n):
        return self.msgs[n]

    # implement
    def add_msg(self, m):
        return

    def add_msg_and_index(self, m):
        self.add_msg(m)
        line_at = self.total_msgs - 1
        self.indexing(m, line_at)

    # implement
    def indexing(self, m, l):
        return
```

An instance of this class can

- store a message in `self.msgs` (by `add_msg()`)
  - `self.msgs` is a list of string, storing all messages.
- index the words in the message in `self.index` (by `indexing()`)
  - `self.index` is a dictionary, whose keys are the words and the values are lists whose elements are the line indices (`line_at`) of the messages that contain the word.
- store a message and index the words (by `add_msg_and_index()`)

In `add_msg_and_index()`, we call `add_msg()` and `indexing()` to store and index the message. Think about the role of `self.total_msgs`; we use it to index the message.

Please read the specifications of UP1 for details.

Continue: `Index` class

```
# implement: query interface
def search(self, term):
    '''
    return a list of tupple. if index the first sonnet (p1.txt),
    call this function with term 'thy' will return the following
    [(7, " Feed'st thy light's flame with self-substantial fuel,
     (9, ' Thy self thy foe, to thy sweet self too cruel:'),
     (9, ' Thy self thy foe, to thy sweet self too cruel:'),
     (12, ' Within thine own bud buriest thy content,')]
    '''
    msgs = []
    return msgs
```

- `search()`: the main query interface. Given a term (i.e., a word), it returns the messages that contains the word. (See the specification in the figure above.)

Continue: `Index` class

We should test the Index class by creating an instance and feeding arbitrary text to it. For example,

```
In [10]: my_idx = Index("Adam")

In [11]: my_idx.add_msg_and_index("Hello world!")

In [12]: my_idx.add_msg_and_index("Welcome come to the west world")

In [13]: my_idx.search("world")
Out[13]: [(1, 'Welcome come to the west world')]
```

You may notice that here “world!” and “world” are treated as different words. This is a bug! (You can handle it later.)

PIndex class

This class is for indexing the poems. It is a child class of Index class and has some special attributes and methods for indexing the sonnets.

```
class PIndex(Index):
    def __init__(self, name):
        super().__init__(name)
        roman_int_f = open('roman.txt.pk', 'rb')
        self.int2roman = pickle.load(roman_int_f)
        roman_int_f.close()
        self.load_poems()

        # implement: 1) open the file for read, then call
        # the base class's add_msg_and_index
    def load_poems(self):
        return

        # implement: p is an integer, get_poem(1) returns a list,
        # each item is one line of the 1st sonnet
    def get_poem(self, p):
        poem = []
        return poem
```

Note: The file is given, which is actually generated by the roman2num.py.

When the instance of PIndex is initialized, it loads the poems from the specified file. (see the example in the next page)

Continue: `PIndex` class

```
In [33]: sonnets = PIndex("AllSonnets.txt")

In [34]: sonnets.get_poem(2)
Out[34]:
['II.',
 '',
 'When forty winters shall besiege thy brow,',
 " And dig deep trenches in thy beauty's field,",
 " Thy youth's proud livery so gazed on now,",
 " Will be a totter'd weed of small worth held:",
 ' Then being asked, where all thy beauty lies,',
 ' Where all the treasure of thy lusty days;',
 ' To say, within thine own deep sunken eyes,',
 ' Were an all-eating shame, and thriftless praise.',
 " How much more praise deserv'd thy beauty's use,",
 " If thou couldst answer 'This fair child of mine",
 " Shall sum my count, and make my old excuse,'",
 ' Proving his beauty by succession thine!',
 ' This were to be new made when thou art old,',
 " And see thy blood warm when thou feel'st it cold.",
 '',
 '',
 '']
```

Initialization of a instance of PIndex class.

Obtain a sonnet by its corresponding Roman number.

PIndex class

This class is for indexing the poems. It is a child class of Index class and has some special attributes and methods for indexing the sonnets.

```
class PIndex(Index):
    def __init__(self, name):
        super().__init__(name)
        roman_int_f = open('roman.txt.pk', 'rb')
        self.int2roman = pickle.load(roman_int_f)
        roman_int_f.close()
        self.load_poems()

        # implement: 1) open the file for read, then call
        # the base class's add_msg_and_index
    def load_poems(self):
        return

        # implement: p is an integer, get_poem(1) returns a list,
        # each item is one line of the 1st sonnet
    def get_poem(self, p):
        poem = []
        return poem
```

You need to open the file that contains all sonnets, loading them line by line, and indexing them by calling the inherited method add_msg_and_index().

Hint: you need to convert the integer to Roman number. Finding the Roman number in the self.msgs, then, loading the poem.

Possible improvements

There are a few things you can improve.

- “world!” is treated as a word (not “world”) to be indexed.
- It does not detect duplicates.

```
In [39]: sonnets.search("can")
Out[39]:
[(226, "   And nothing 'gainst Time's scythe can make defence"),
 (256, ' Nor can I fortune to brief minutes tell,'),
 (340, ' So long as men can breathe, or eyes can see,'),
 (340, ' So long as men can breathe, or eyes can see,'),
```

- Can we search for phrases, e.g., “hello world”?

Please read the specification of UP1 for more details!

Terminal commands

- All computers nowadays have GUIs
  - GUI = Graphical User Interface
  - But not so in early days
  - Terminal/CMD was the only way to manage your file system

- The chat system is designed to handle multiple clients
  - Hence we need multiple sockets
  - Python’s shell does not support such purpose
  - But Terminal/CMD does

Mac OS X Terminal

To open the terminal on your mac:

- The terminal is an app, you can find it in the Launchpad/Other
- Or, you can press “command + space”, then, input “terminal” to find it.

You can change the appearance of the terminal window in the “Preferences”

The basics

The shell's (i.e., terminal's) grammar:

- Generally, it is a **command** followed by **arguments**, such as the path of a file, or, some parameters
  - E.g., **cd** the_path changes the current directory to the path specified

```
NYUSH1742LP-MX:~ xg7$ cd Documents/Teaching/
NYUSH1742LP-MX:Teaching xg7$
```

  - E.g., **ls,** it lists the files and folders in the current directory. It has several optional arguments, such as -l.

```
NYUSH1742LP-MX:Movies xg7$ ls -l
total 0
drwxr-xr-x  2 xg7  staff   64 Nov 21  2018 ApowerREC
drwxr-xr-x  4 xg7  staff  128 Oct 19 09:23 Videos
NYUSH1742LP-MX:Movies xg7$
```

- You can use the ↑↓ to find the history of your inputs quickly.

Some Mac OS X commands

- **cd** the_path: it changes the current directory to the one specified by the_path.
  - in the shell, “.” represents the current directory, and “..” represents the parent directory of the current directory. So, cd .. changes the current directory to the upper level.
  - “~”: represents the home directory. cd ~ (or just cd) changes the current directory to the home of the current user.
  - “/”: represents the root directory. cd / changes the current directory to the root. (Caution: Don’t change any files or folders there if you don’t know what you are doing.)

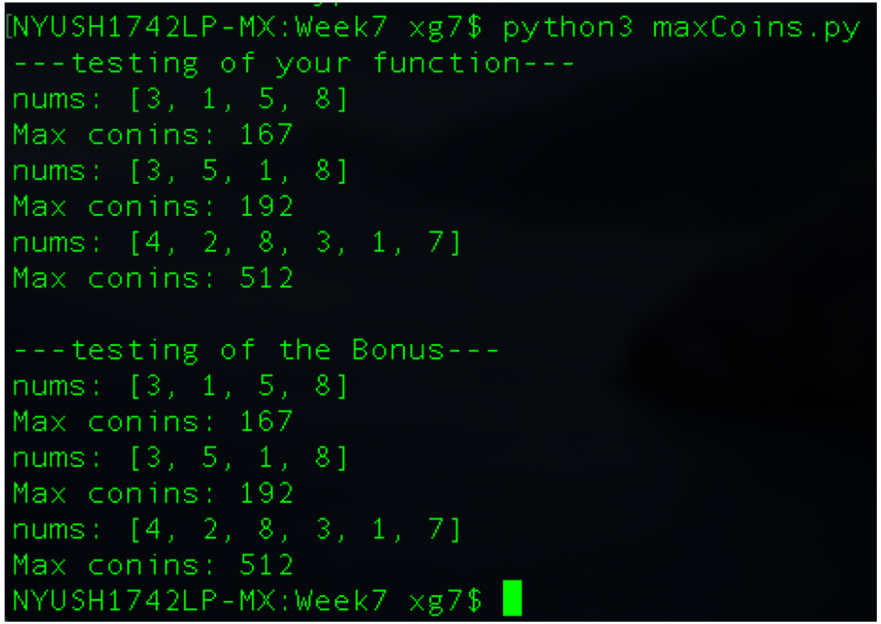
Some Mac OS X commands

- ls [options] [file|dir]: it lists everything in the current directory.
  - Some optional arguments:
    - -l: list with long format, show permissions.
    - -a: list all files including the hidden files
    - -s: list file size
    - -S: sort by file size
    - -t: sort by time & date

Some Mac OS X commands

- python3 [path+file_name]: run the file specified using Python3
  - If no path specified, it runs the file (specified by file_name) in the current directory.

Note: usually, calling python [path+file_name] runs the file with Python2. Actually, you can change the setting so that calling python runs Python 3.



Basic Windows cmd commands

- cd the_path: it changes the current directory to the one specified by the_path.
  - in the cmd, “.” represents the current directory, and “..” represents the parent directory of the current directory. So, cd .. changes the current directory to the upper level.
  - “\”: represents the root directory. cd / changes the current directory to the root. (Caution: Don’t change any files or folders there if you don’t know what you are doing.)

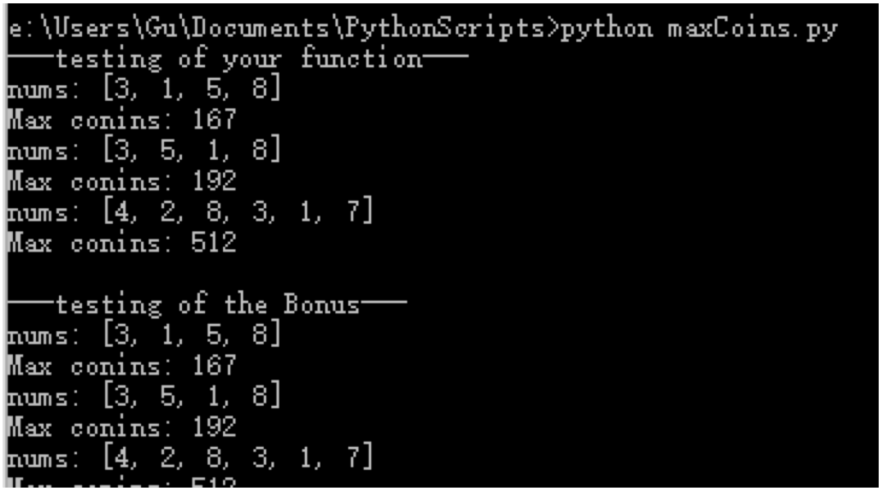
Basic Windows cmd commands

- dir [options]: it lists everything in the current directory.
  - Some optional arguments:
    - /a: list all files including the hidden files
    - /w: list the directories

Some Windows cmd commands

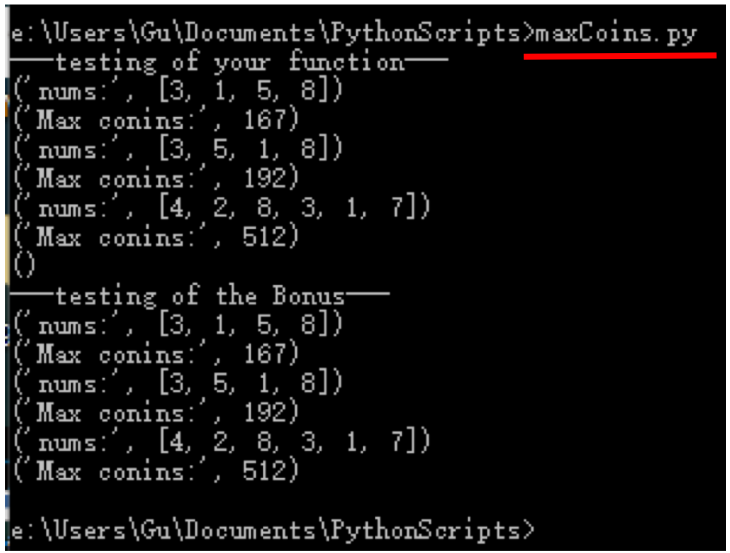
- python [path+file_name]: run the file specified using the Python you installed. (Note: we use Python 3 in our course)
  - If no path specified, it runs the file (specified by file_name) in the current directory.

Note: if you have Python installed, you'd better to add the path of folder that the Python installed to the Path in the environment variables. Then, you can call python in any directory in the cmd.



Some Windows cmd commands

- Actually, Python files can run directly in cmd.
- Just input the name of the .py file in the cmd, then, it runs.



Reference

Mac OS X terminal command cheat sheet:

https://gist.github.com/poopsplat/7195274

Windows cmd command cheat sheet (referring to Unix):

https://gist.github.com/jonlabelle/e8ba94cd29b8f63fd7dd3c4f95c1d210

By the way, it is not necessary to recite those commands on purpose. You will get familiar with them if you use them frequently.