Basic Python Programming

Yilin Zheng

Dept. of Computer Science and Engineering

July 9, 2017

Outline I

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Introduce to OOP

- OOP is **Object-Oriented Programming**
- Differed from OOP, using **functions(blocks) to design programs** is process oriented
- In OOP, we wrap data and functions together to design programs in more abstract way

OOP Terminology

- Class: A user-defined **prototype** for an object that defines a set of **attributes** that characterize any object of the class. The attributes are **data members (class variables and instance variables)** and **methods**, accessed via **dot notation**
- Class variable: A variable which is **shared by all instances of a class**(can be accessed). Class variables are **defined within a class but outside any of the class's methods**. Class variables are not used as frequently as instance variables are
- Data member: A **class variable** or **instance variable** that holds **data associated** with a class and its objects
- Method : **A special kind of function** that is defined in a class definition

OOP Terminology

- Object: **A unique instance of a data structure** that's defined by its class. An object comprises both **data members (class variables and instance variables)** and **methods**
- Instance: An individual object of a certain class. An object belongs to a class, which is an instance of certain
- Instance variable: A variable that is defined **inside a method** and belongs **only to the current instance** of a class
- Inheritance: The **transfer** of the characteristics of a class to other classes that are derived from it

Features

- Features for OOP: **Encapsulation**, **Inheritance**, **Polymorphism**
- **Encapsulation** means a language construct that facilitates the bundling of data with the methods (or other functions) operating on that data
- **Inheritance** is when an object or class is based on another object (prototypal inheritance) or class (class-based inheritance), using the same implementation (inheriting from an object or class: inheriting behavior, programming by difference) or specifying a new implementation to maintain the same behavior (realizing an interface)
- **Polymorphism** is the provision of a single interface to entities of different types

self

- Use `self` to indicate the **instance** itself
- Note: Using 'self' rather than other words like'this'(C++) is not peremptory rule but is a conventional rule

Class

- Form: `class Class_name(object):`(Python2) or `class class_name():`(Python3)
- In block, we can either define **class variables** or **methods**
- Directly use an instance name with assign operator `=` and the class
- Don't forget to use `self` **as first parameter of methods**

Method

- Functions defined within classes is called methods
- The only different between methods and functions is that methods always take **self as first and essential parameter**
- When an instance is created, the **methods and variables** will all be **inherited**

__init__

- `__init__` will firstly be executed when an instance is created
- Use this method to initialize some variables or execute some codes

Class variable

- Class variables are defined **within class**
- They are **bound with the namespaces of the class** and only valid **under the premise of its namespaces**
- They can be accessed by **all its instances**
- There is only **one copy** of the class variable so when any one object makes a change to a class variable, that change will **be seen all the other instances**
- Use **class name** with **dot notation** to refer to class variables

Instance variable

- Instance variables are defined **within method**
- Use `self` and **dot notation** to create and refer to
- They are **unique to each instance**, **bound with the namespaces of the object**
- Each object has **its own copy of the field** so they are not shared and not related to the same name in a different instance

Inherit

- In inheritance, a class can inherit from other class. The former is called **subclass** or **derived class**, the latter is called **superclass** or **base class**. **Subclasses inherit from superclasses**
- If Class B inherits from Class A, then the change taking place in Class A will **automatically reflected** in Class B, inversely, the change in Class B will **take no effect** in Class A.
- Form: `class B(A): #B inherit from A`
- A subclass can inherit from **one or multiple superclass(es)**, called **multiple inheritance**
- Form: `class B(A, C, D, E...):#B inherit from A, C, D, E...`
- We can refer to subclasses through superclasses, which is called **polymorphism**

Input

- Get input from I/O devices like keyboards, later we will learn how to input from files or storages
- Use:
  - `input(XXXX)` (Python2 and 3)
  - `raw_input(XXXX)` (Python2)
- XXXX is prompt for users, usually we use strings like 'Enter an integer:'

input

- `input` can be use both in **Python2 and 3**
  - In Python2, it only **reads numbers** and **return numbers**

```
>>> input('Enter a number:') #Python 2
Enter a number:10.5
10.5
>>> input('Enter a number:') #can not
                             read
                             strings
Enter a number:number
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'number' is not
                             defined
```

input

- In Python3, it reads both **numbers** and **strings** and **return only in string format**
- `int` or `float` to get matching format

```
>>> input('Enter a number:')
Enter a number:250
'250'
>>> int(input('Enter a number:')) #
                                change to
                                int
Enter a number:10
10
>>> float(input('Enter a number:')) #
                                change to
                                float
Enter a number:10.5
10.5
```

raw_input

- `raw_input` can only be used in **Python2**, it reads both **numbers** and **strings** and **return only in string format**
- Also can use `int` or `float` to get matching format

```
>>> raw_input('Enter a number:')
Enter a number:10
'10'
>>> float(raw_input('Enter a number:'))
Enter a number:233333.33
233333.33
```

Multiple data input

- If we want to input multiple data in a time, we can use `eval`
- The data input should be separated by **comma**

```
>>> eval(input('Enter three number:'))
Enter three number:1,2,3
(1, 2, 3)
>>> a, b, c = eval(input('Enter three number:'
                         ))
Enter three number:1,2,3
>>> a
1
>>> b
2
>>> c
3
```

Output

- Use `print` to output what we want
- Formatted output: format specifier
  - %d — decimal
  - %x — hexadecimal
  - %o — octal
  - %f — float
  - %s — string
  - %c — characters
  - More on ▸ Format Specification
- Use **format specifiers** to output data in corresponding format
- And use `%(Variables)` to place the value into the result

```
>>> age = 20
>>> print('My age is %d'%(age)) #put at the
                                 end of string
My age is 20
```

Formatted output I

- The whole %: %[name][flag][width][.][precision]type
  - **name** can be empty, numbers(occupy), key of dict
  - **flag** is format qualifier: +(right alignment), -(left alignment), #(pad 0 for oct, pad 0x for hex), **0**(zero)(pad 0)
  - **width** is to control the max length
  - **precision** is to control the decimal
  - **type** is to indicate the type of the data

```
>>> print("%-7s3" %("Python"))
Python 3
>>> print("%(when)s is %(year)d" % {"when":"
                                this year","year":
                                2017})
this year is 2017
>>> print("%.5f" %3.1415926535)
3.14159
>>> print("%.4e" %20170708) #output 2.0171e+07
```

Formatted output II

- Another way to do formatted output is using `format`
- The whole `format`:
  {[name][:][[fill]align][sign][][0][width][,][.precision][type]}
  - Need to use **braces** to pass name such as **keys of dict**
  - **fill** can use any characters
  - **align** is format qualifier: >(right alignment), <(left alignment), ^(center alignment), =(length)
  - **sign** means positive(+) or negative(-)
  - Rests are the same with % format

```
>>> print('{:*=8}'.format(2000))
****2000
>>> print("{site[0]}.{site[1]}.{site[2]}".
                                format(site=["www",
                                "google","com"]))
www.google.com
>>> print("{:,}".format(123456)) # 123,456
```

File

- We often need to manage files, so how to read and write files by Python is concernful
- Use built-in function `open` to open a file and create an object of class **file**
- File objects contain **methods** and **attributes** that can be used to collect information about the file we opened
- We can open files in different **modes**
  - `'r'` — **read** mode, which is default mode if omitted; read files only
  - `'w'` — **write** mode; edit and write new information to the file (**any existing files with the same name will be erased when this mode is activated**)
  - `'a'` — **appending** mode; add new data to the end of the file
  - `'r+'` — **read&write** mode; handle both read and wirte actions
  - More on ▸ Modes

Example: file_1

- open need two parameters(**strings**): **file path** and **mode**
- Form: file_object_name =open(file_path, mode)

```
>>> file_1 = open('/Users/jerry/Desktop/data',
                  'r')
>>> file_1
<_io.TextIOWrapper name='/Users/jerry/Desktop/
    data' mode='r' encoding
    ='UTF-8'>
```

Operations on files

- There are some common operations on files: `read`, readline, `readlines`, `write`, `writelines`, `seek`, `close`
  - **read** can return the content in the file in **string** format
  - **readline** reads **one line each time**
  - **readlines** can read **whole file and analyse to lines**
  - **write** can write **strings** to file
  - **writelines** can write multiple lines to file but need to **add newline(`n`) manually**
  - **seek** can **move pointer to certain position** of file
  - **close** is to **close files** after completng operations, it's vital!!!

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Example I

- Create a file named 'data' and write something in it

```
>>> file_1 = open('/Users/jerry/Desktop/data',
                                    'r+')
>>> file_1.read() #read
'Hello~~~'
>>> file_1.seek(0, 0)#pointer goes back to
                          beginning
0 #first number is offset, second one is from
                          where
>>> file_1.readline()#read a line
'Hello~~~'
>>> file_1.write('Hi, this is a stirng')#write
20 #the number of chars writen in
>>> file_1.seek(0)#rewind
0
>>> file_1.readlines()
['Hello~~~Hi, this is a stirng']
```

Example II

```
>>> strings = ['\nthis is 2nd line\n', '3rd
                            line']
>>> file_1.writelines(strings)#write multiple
                            lines
>>> file_1.seek(0)
0
>>> file_1.readlines()
['Hello~~~Hi, this is a stirng\n', 'this is
                          2nd line\n', '3rd line'
                          ]
>>> file_1.close();#done, remember to close it
```

pickle

- Use module **pickle**, we can **store** and **load objects** in files, which is called storing the object persistently
- Use `pick.dump` to store data:
  `pickle.dump(obj, file, [,protocol])`
  - **obj** is the objects to store in
  - **file** is the storage location, a file object
  - **protocol** is to indicate how to serialize, which is an integer of protocol version
    - **0**: the **original ASCII protocol** and is backwards compatible with earlier versions of Python
    - **1**: the **old binary format** which is also compatible with earlier versions of Python
    - **2**: introduced in Python 2.3, it provides much more efficient pickling of new-style classes

pickle

- **3**: added in Python 3.0, it has explicit support for bytes objects and cannot be unpickled by Python 2.x. This is **the default protocol**, and the recommended protocol when compatibility with other Python 3 versions is required
- **4**: added in Python 3.4. It adds support for **very large objects**, pickling more kinds of objects, and some data format optimizations

```
#!/usr/bin/python # Filename: use_pickle.py
import pickle #import the module
data_1 = {'a': [1, 2.0, 3, 4+5j], 'b': ('
                        String'), 'c': 1.314}
file_1 = open('data.pkl', 'wb')
pickle.dump(data_1, file_1, 3)#protocol 3
file_1.close()#close file
del(data_1, file_1) #delete data and file
                        pointer
```

pickle

- Use `pickle.load` to get objects: `pickle.load(file)`

```
#continue the script
file_2 = open('data.pkl', 'rb')
data_2 = pickle.load(file_2)#load data
print(data_2)
```

- It will print the outcome

```
{'a': [1, 2.0, 3, (4+5j)], 'b': 'String', 'c':
                                    1.314}
```