Basic Python Programming

Yilin Zheng

Dept. of Computer Science and Engineering

July 6, 2017

Outline I

1. Review
   - Data structure
   - List
   - Tuple
   - Dictionary
   - Sequence
   - Set
   - Reference
   - String
2. Object-Oriented Programming
   - OOP
   - Self
   - Method

Outline II

- __init__
- Variable
- Inherit

Data structure

- Data structure is **a particular way of organizing data**
- Built-in data structures: **list**, **tuple**, **dictionary**, **Set**

List

- Items are enclosed by **square brackets** [ ]
- The items can be **numbers, characters, strings and so on**
- **All the items should be the same type** and **seperated** by **comma ,**
- Use **index operator** to refer to the items in a list

Operations on list

- Functions and methods: `len`, `sort`, `sorted`, `del`, `append`
- Iteration on items

Tuple

- Tuple's **items cannot be changed**
- Use **parenthesis** ( )to enclose items

Operations on Tuple

- functions like `len`, `sorted`
- Iteration on the items

Dictionary

- For dictionary, items are **enclosed by braces**
- Every item consists of two parts: **key** and **value**, seperated by **colon** :
- **Key** should be **unique** and **cannot be changed**
- **Value** has **no these limitations**
- **Refer to value by key**

Operations on dictionary

- functions: `len`, `sorted`, `del`
- Iterations on items

Sequence

- Sequence: **List**, **tuple** and **dictionary**
- Features: **Index operator** and **Slice operator**
- Use index operator to **find a specific item**
- Use slice operator to **get a part of sequence**
- Index operator: `[index]`
- Slice operator: `[start:end]` or `[start:end:step]`
- If step >0, from left to right; if step $<$ 0, from right to left (**cannot be zero**)

Slice operator

- if negative, it will take out the slice **in the opposite**

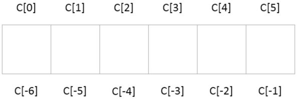

Figure: sequence

Set

- Sets are **unordered collections**
- Use `set` to create sets

Operations on set

- functions and methods: `len`, `add`, `remove`, `copy`
- Operator: `in`

Reference

- When you create an object and assign a varible to it, the variable only **refers to** the object and doesn't represent the object itself
- The variable name points to the part of memory the object stored, which is called **binding the name to the object**

More on string

- All the strings is the objects of class **str**
- Some methods of strings: `len`, `startswith`, `in`, `find`, `join`
- More on ▸ String Methods

Introduce to OOP

- OOP is **Object-Oriented Programming**
- Python has been an object-oriented language since it existed
- 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
- Hold on, please!!!

Figure: Warning

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
- Operator overloading: The assignment of **more than one function** to a particular **operator**
- Function overriding: The behavior of child class **inheriting every single functionality** from parent class

Features

- There are some significant 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

Don’t lose your heart

- Emmmmmmmm......Nothing happened

self

- Use `self` to indicate the **instance** itself

```
>>> class what_is_self(object): #define a
                           class
...     def print_self(self):
...         print(self)
...         print(self.__class__)
...
>>> test = what_is_self()#create an instance
>>> test.print_self() #call the method
<__main__.what_is_Self object at 0x10bd1b208>
                           #self
<class '__main__.what_is_Self'> #self.
                           __class__
```

- Note: Using 'self' rather than other words like'this'(C++) is not peremptory rule but is a conventional rule

Class

- Previous page has simply shown how to define a class inherited from `object` and create instances of it
- Use keyword `class` with a class name and parameter `object` that enclosed by parenthesis, ended with **colon** to begin the definition, which is `class Class_name(object):` at the first line
- With an indentation ahead, we can either define **class variables** or **methods** for the class
- After that, if we want to create an instance of the class, directly use an instance name with assign operator `=` and the class, like assgining a value to an variable
- Don't forget to use `self` **as first parameter of methods**

Example: Student (based on object)

- Why need object? Because object is **base class**
- Note: In Python3, we don't need to write object since there is no difference

```
#!/usr/bin/python # Filename: class_def.py
class Student:
    count = 0 #class variable
    books = {} #empty dict
    #the following function is called
                               constructor
    def __init__(self, name, age):#self is
                               always the
                               first parameter
        self.name = name #use dot notation to
                                   refer to
        self.age = age
    pass
```

Example: Undergraduate(based on Student)

- If we want to define a class inherited from certain class XXX, we should replace `object` with XXX

```
class Undergraduate(Student):
#this class will have all its superclass has
def add_student(self):
    self.roster[self.name] = self.age
def print_roster(self):
    print(self.roster)
Student_1 = Undergraduate(’Tom’, 20)
Student_1.add_student() #add the detail to
                            dict
Student_1.print_roster()#print the dict
```

Method

- Class can have defined functions as 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**

```
#!/usr/bin/python # Filename: method_def.py
class Person:
    def sayHi(self): #method sayHi
        print(’Hello, how are you?’)
    def ansHi(self): #method to answer
        print(’Emmmmmmm’)
p1 = Person()#person_1
p1.sayHi()
p2 = Person()#person_2
p2.ansHi()
```

__init__

- There are many names having special meaning: `__init__`, `__le__`, `__getattribute__`, `__new__`
- `__init__` will firstly be executed when an instance is created

```
#!/usr/bin/python # Filename: class_init.py
class Person:
    def __init__(self, name):
        self.name = name
        print(’Instance created’)
    def sayHi(self):
        print(’Hi, my name is’, self.name)
p1 = Person(’Optimus Prime’)
p1.sayHi()
```

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 willtextbfbe 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

Example: Count_Bit_1

```
#!/usr/bin/python # Filename: class_def.py
class Count_Bit_1:
    count = 0 #class variable
    def __init__(self, n):
        self.n = n; #instance variable
        print(’n is %d’%(self.n))
    def count_bit_1(self):
        while self.n!=0:
            Count_Bit_1.count += (self.n&1)
            self.n = self.n>>1
        print("There is %d bits of 1"%(
                                   Count_Bit_1
                                   .count))
test = Count_Bit_1(int(input("Enter an integer
                           : ")))
test.count_bit_1()
```

Inherit

- Inheritance is one of the major benefits of OOP, which can reuse code, easy to maintain and save time
- 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. That is **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
```

- So, we can save codes for comman features and characterize the unique part
- Also, we can refer to subclasses through superclasses, which is called **polymorphism**

Example

- Define a base class: SchoolMember

```
#!/usr/bin/python # Filename: class_inherit.py
class SchoolMember:
'''Represents any school member.'''
def __init__(self, name, age):
    self.name = name
    self.age = age
    print('(Initialized SchoolMember: %s)'%(
                               self.name))
#tell is to print the name and age of the
                           person
def tell(self):
    '''Tell my details.'''
    print('Name: %s Age:"%d"'%(self.name, self
                               .age))
```

Example

- Define a subclass: Teacher, it has additional feature **salary**

```
class Teacher(SchoolMember):
'''Represents a teacher.'''
def __init__(self, name, age, salary):
    SchoolMember.__init__(self, name, age) #
                               refer to the
                               constructor of
                               superclass
    self.salary = salary #Attention
    print('(Initialized Teacher: %s)'%(self.
                               name))
def tell(self):
    SchoolMember.tell(self) #refer to methods
                               of superclass
    print('Salary: "%d"'%(self.salary))
```

Example

- Define another subclass: Student, it has unique character **mark**

```
class Student(SchoolMember):
'''Represents a student.'''
def __init__(self, name, age, marks):
    SchoolMember.__init__(self, name, age)
    self.marks = marks
    print('(Initialized Student: %s)'%(self.
                               name))
def tell(self):
    SchoolMember.tell(self)
    print('Marks: "%d"'%(self.marks))
```

Example

- Instantiation

```
#instantiation
t = Teacher(’Mr. Smith’, 40, 50000)
s = Student(’Thompson’, 20, 100)
members = [t, s]
for member in members:
    member.tell() #print the detail of s and t
```