Python Programming

Classes

Prof. Chang-Chieh Cheng

Information Technology Service Center

National Yang Ming Chiao Tung University

What is a class?

- In Python, we can define a new data type for some purposes
  - For example, a data type can record all information about a student
    - ID
    - name
    - age
    - scores
- We also can design some functions to manipulate data of a class
- We call such user-defined data type as a **class**
- **class is a data type**
- **An object is an instance of a class**
- **Object-oriented programming (OOP)**
  - **Everything is an object**

Defining a class

- The syntax to define a simple class

```
class class_name:
    data_initial_statements
```

  - The data of a class called **member data**

```
class student: # design a class, datatype
    id = ''
    name = ''
    age = 0
    scores = []

James = student()      # object instancing
James.id = 'A123'
James.name = 'James'
James.age = 18
James.scores = [90.0, 100.0, 85.0]
print(James.id)
print(James.name)
print(James.age)
print(James.scores)
```

Defining a class

- Define a class with methods

```
class class_name:
    # data initialization
    # method definitions
```

- Notice that, a method must has **`self`** parameter to access the member data

```
class student:
    id = ''
    name = ''
    age = 0
    scores = []
    def avg(self):
        return sum(self.scores) / len(self.scores)

James = student()      # object creation
James.id = 'A123'
James.name = 'James'
James.age = '18'
James.scores = [90.0, 100.0, 85.0]
print(James.avg())     # the argument of self is James
```

Defining a class

- Calling a method in a method

```
class student:
    id = ''
    name = ''
    age = 0
    scores = []
    def avg(self):
        return sum(self.scores) / len(self.scores)

    def grade(self, symbols = ['A', 'B', 'C', 'D']):
        a = self.avg()
        if a >= 90.0:
            return symbols[0]
        elif a >= 80.0:
            return symbols[1]
        elif a >= 60.0:
            return symbols[2]
        else:
            return symbols[3]
```

Defining a class

- Calling a method with arguments

```
James = student()      # object creation

James.id = 'A123'
James.name = 'James'
James.age = '18'
James.scores = [80.0, 60.0, 85.0]

print(James.avg())
print(James.grade())
print(James.grade(['Execellent', 'Good', 'OK', 'Fail']))
```

Defining a class

- The definition of a class can be put in a module
  - For example, put `student` class in `School.py`

```
import School as SC
James = SC.student()   # object creation

James.id = 'A123'
James.name = 'James'
James.age = '18'
James.scores = [80.0, 60.0, 85.0]

print(James.avg())
print(James.grade())
print(James.grade(['Execellent', 'Good', 'OK', 'Fail']))
```

Initialization method

- **`__init__(self)`**
  - Initialing the member variables of an object.

Two underlines

- Other special methods for designing a class
  - https://docs.python.org/3/reference/datamodel.html

__init__

- Initializing an object
  - Although you can declare and initialize all member data at any place in a class, **you should do that in `__init__`**

```
class student:
    def __init__(self, id = '', name = '',
                 age = 18, scores = [0.0, 0.0, 0.0]):
        self.id = id
        self.name = name
        self.age = age
        self.scores = scores
```

```
class student:
    id = ''
    name = ''
    age = 0
    scores = []
```

```
Mary = student('A000', 'Mary', 20, [90, 80, 75])
James = student(name = 'James', id = 'A001')
L = [Mary, Jamse]
print(Mary)    # call __str__()
print(James)   # call __str__()
print(L)       # call __repr__()
```

__str__ & __repr__

- Output an object as a string

```
class student:
    ...
    def __str__(self):
        return 'ID: ' + self.id + '\n'\
               + 'Name: ' + self.name +'\n'\
               + 'Age: ' + str(self.age) + '\n'\
               + str(self.scores) + '\n'
    # For printing from a data container
    def __repr__(self):
        return self.__str__()
```

```
James = student()
James.id = 'A123'
James.name = 'James'
James.age = '18'
James.scores = [00.0, 10.0, 85.0]
print(James)
s = str(James)
print(s)
```

Arithmetic Operators

- `__add__(self, other)`
  - self + other
- `__sub__(self, other)`
  - self - other
- `__mul__(self, other)`
  - self * other
- `__truediv__(self, other)`
  - self / other➔real number division
- `__floordiv__(self, other)`
  - self // other ➔integer division

Arithmetic Operators

```
class student:
    ...
    def __add__(self, scores):
        temp = student(self.id, self.name, self.age, self.scores)
        for i in range(len(temp.scores)):
            temp.scores[i] += scores[i]
        return temp
    def __truediv__(self, div):
        temp = student(self.id, self.name, self.age, self.scores)
        for i in range(len(temp.scores)):
            temp.scores[i] /= div
        return temp
    def __floordiv__(self, div):
        temp = student(self.id, self.name, self.age, self.scores)
        for i in range(len(temp.scores)):
            temp.scores[i] //= div
        return temp
```

Copying self

Returning the copy

```
James1 = student('A001', 'James', 19, [60, 40, 35])
James2 = James1 + [1.0, 2.0, 5.0]
James3 = James1 / 2
James4 = James1 // 2
```

Arithmetic Operators

- `__iadd__(self, other)`
  - self += other ➔ self = self + other
- `__isub__(self, other)`
  - self -= other
- `__imul__(self, other)`
  - self *= other
- `__itruediv__(self, other)`
  - self /= other
- `__ifloordiv__(self, other)`
  - self //= other

Arithmetic Operators

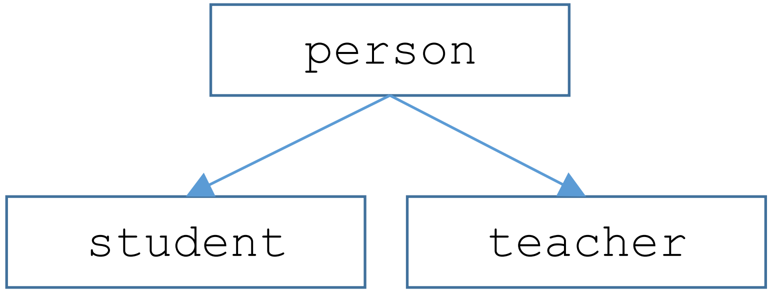
```
class student:
    ...
    def __iadd__(self, scores):
        for i in range(len(self.scores)):
            self.scores[i] += scores[i]
        return self               # Must return self!
```

```
James1 = student('A001', 'James', 19, [60, 40, 35])
James1 += [1.0, 2.0, 5.0]
```

Inheritance

- Person

```
class person:
    def __init__(self, id = '', name = '', age = 0):
        print('person.__init__')
        self.id = id
        self.name = name
        self.age = age
    def __str__(self):
        return 'ID: ' + self.id + '\n'\
                + 'Name: ' + self.name +'\n'\
                + 'Age: ' + str(self.age)
    def __repr__(self):
        return self.__str__()
```



Inheritance

- Student

```
class student(person):
    def __init__(self, id = '', name = '', age = 0, scores = []):
        print('student.__init__')
        super().__init__(id, name, age)
        # Alternatively, person.__init__(self, id, name, age)
        self.scores = scores
    def avg(self):
        return sum(self.scores) / len(self.scores)
```

- Teacher

```
class teacher(person):
    def __init__(self, id = '', name = '', age = 0, title = 'Prof.'):
        print('teacher.__init__')
        super().__init__(id, name, age)
        self.title = title
    def __str__(self):
        return 'ID: ' + self.id + '\n'\
              + 'Name: ' + self.title + ' ' + self.name +'\n'\
               + 'Age: ' + str(self.age)
```

Inheritance

- Instancing the classes

```
J = student(id = '0088', name = 'James', age = 20, scores = [100, 50])
print(J)
print(J.avg())

T = teacher(id = 'T001', name = 'Cheng', age = 50)
print(T)
```

Inheritance

- Student

```
class student(person):
    def __init__(self, id = '', name = '', age = 0, scores = []):
        print('student.__init__')
        person.__init__(self, id, name, age) # DO NOT use super()!
        self.scores = scores
    def avg(self):
        return sum(self.scores) / len(self.scores)
```

- Teacher

```
class teacher(person):
    def __init__(self, id = '', name = '', age = 0, title = 'Prof.'):
        print('teacher.__init__')
        person.__init__(self, id, name, age) # DO NOT use super()!
        self.title = title
    def __str__(self):
        return 'ID: ' + self.id + '\n'\
              + 'Name: ' + self.title + ' ' + self.name +'\n'\
               + 'Age: ' + str(self.age)
```

Exercise 1.1

- Design a class named `lineseg` to describe a 2D line segment:
  - `lineseg ( x1 = 0, y1 = 0, x2 = 0, y2 = 0)`
    - `(x1, y1)` indicates the start point of a line segment;
    - `(x2, y2)` indicates the terminal point of a line segment.
  - `lineseg.sumsq()` returns the sum of square differences by$(x_2 - x_1)^2 + (y_2 - y_1)^2$.
  - `lineseg.length()` returns the length of a line segment by$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.
  - An object of `lineseg` can be multiplied by a float number.
- Your `lineseg` must satisfy the following test code:

```
line = lineseg (1., 2., 4., 6.)
print(line.x1, line.y1, line.x2, line.y2, line.sumsq(), line.length())
line *= 2.0
print(line.x1, line.y1, line.x2, line.y2, line.sumsq(), line.length())
```

- The results will be:

```
1.0, 2.0, 4.0, 6.0, 25.0, 5.0

1.0, 2.0, 8.0, 12.0, 149.0, 12.2066
```

- Notice that the error must be in ±`0.001`.

Exercise 1.2

- Design a class `lineseg3` inherited from `lineseg` to describe a 3D line segment.
  - `lineseg3 ( x1 = 0, y1 = 0 , z1 = 0, x2 = 0, y2 = 0, z2 = 0)`
    - `(x1, y1, z1)` indicates the start point of a line segment;
    - `(x2, y2, z2)` indicates the terminal point of a line segment.
  - `lineseg3.sumsq()` returns the sum of square differences by $(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$.
  - `lineseg3.length()` returns the length of a line segment by $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$.
  - An object of `lineseg3` can be multiplied by a float number.
- Your `lineseg3` must satisfy the following test code:

```
line = lineseg3 (1., 2., 3., -1., -2., -3.)
print(line.x1, line.y1, line.x2, line.y2, line.sumsq(), line.length())
line *= 2.0
print(line.x1, line.y1, line.x2, line.y2, line.sumsq(), line.length())
```

- The results will be:

```
1.0, 2.0, 3.0, -1.0, -2.0, -3.0, 56.0, 7.483

1.0, 2.0, 3.0, -2.0, -4.0, -6.0, 126.0, 11.225
```

- Notice that the error must be in ±`0.001`.