Python Workshop

7. Classes

Quick jump: Class | Advanced topics

Classes

- Sometimes we want to bundle data (variables) and functions together.
- In this case we can create a class.
- Like functions, we have to define a class first and use it in our code.
- To define a class, we use the keyword `class`, for example:

```
class Hello:
    msg = "How are you?"
    def hello(self):
        print('Hi!', self.msg)
```

Class definition

- We can define variables and functions in a class. In the previous example:

```python
class Hello:
    msg = "How are you?"
    def hello(self):
        print('Hi!', self.msg)
```

- msg is an **attribute** of the class `Hello`, and `hello` is a class **method**.
- A class method must be defined with `self` as the first parameter.

Object instantiation

- Class is a **prototype** of objects. One a class is defined, we can instantiate objects based on the class definition by adding `()` after the class name.
- Access operator `.` is used to access attribute and method in an object.

```
h = Hello()
print(h.msg)
h.hello()
```

Output:

```
How are you?
Hi, How are you?
```

Self-learning topics (~60min)

- Class members
- Constructor
- Naming convention
- Advanced topics:
    - Static & class methods
    - Inheritance
    - Duck typing
    - Errors and Exceptions
    - File IO

Class members

Attributes

- **Attributes** are variables kept in an object. It represents the state of an object.
- For example, we can define a box with 3 dimensions:

```
class Box:
    width = 10
    height = 20
    depth = 5
```

Object instantiation

- When we create an object from a class, the object will owns a new copy of the attributes. So if we create two objects:

```
b1 = Box()
b2 = Box()
```

- `b1` and `b2` will then owns a different sets of attributes `width`, `height` and `depth`.

Accessing attributes

- Once defined, we can access attribute value using the access operator `.`.

```
class Box:
    width = 10
    height = 20
    depth = 5

b1 = Box()
print(b1.width, b1.height, b1.depth)
```

Accessing attributes

- We can modify values of an attributes also.

```
class Box:
    width = 10
    height = 20
    depth = 5

b1 = Box()
b1.depth = 50
print(b1.width, b1.height, b1.depth)
```

Methods

- **Methods** are behaviours of an object.
- Results may be affected by the value of the attributes. For example:

```
class Box:
    width = 10
    height = 20
    depth = 5
    def getVolume(self):
        return self.width * self.height * self.depth
```

- The first parameter of a method must be `self`, which is explaned on the next page.

self

- With self, the method can return results base on the values in the object.

```
b1 = Box()
b2 = Box()
b2.depth = 50
print(b1.getVolume())
print(b2.getVolume())
```

- Note that getVolume() return the volume according to the attribute values in b1 and b2.

Constructor

Attribute creation in method

- Instead of defining attributes in a class, we can also create new attributes in a method by simple **assignment:**

```
class Box:
    def setDimension(self, width, height, depth):
      self.width = width
      self.height = height
      self.depth = depth

    def getVolume(self):
        return self.width * self.height * self.depth

b = Box()
b.setDimension(10, 20, 30)
print(b.getVolume())
```

Customizing constructor

- In the previous example, we defined a method `getDimensions()` to initialize attributes of the object.
- In fact, it will be much better if we can initialize the attributes when we create an object, like this:

```
b = Box(10, 20, 30)
print(b.getVolume())
```

- This could be done by defining a special method `__init__()` in the class.
- This method is called the constructor.

init()

- The previous example can then be modified to:

```python
class Box:
    def __init__(self, width, height, depth):
      self.width = width
      self.height = height
      self.depth = depth

    def getVolume(self):
        return self.width * self.height * self.depth

b = Box(10, 20, 30)
print(b.getVolume())
```

More about constructor

- The **constructor** is simply a function definition, so we can define parameter lists in the same way as any other functions.
- For example, we can make use of default values

```
class Box:
    def __init__(self, width = 1, height = 1, depth = 1):
      self.width = width
      self.height = height
      self.depth = depth

    def getVolume(self):
        return self.width * self.height * self.depth

b1 = Box()
b2 = Box(10, 20, 30)
print(b1.getVolume(), b2.getVolume())
```

Naming convention

The underscore

- In Python, underscore `_` is used in a number of places for special purposes.
- For example, any function starting with `_` is considered **hidden** or **private**.
- In fact, if you define a function with name starting with `_`, it will not be imported by `import *`

Underscore example

- Suppose we have a file `myUtil.py` with function `_a()` defined:

```
# myUtil.py
def _a():
  pass
```

Then if we use `import *` to import it in another file, `_a()` cannot be used. This will give an error.

```
from myUtil import *
_a()
```

Underscore in a class

- Attributes and methods starting with `_` has a meaning that they are private, i.e., not to be used outside the class.
- In the following example, `_w`, `_h`, and `_d` are considered private attributes of the class, we should not access them directly **outside** of the class.

```python
class Box:
    def __init__(self, w, h, d):
        self._w, self._h, self._d = w, h, d

    def getVolume(self):
        return self._w * self._h * self._d
```

Convention only

- However, note that the private attribute names are just a convention, they are still accessible.
- The following code still works perfectly, but it is not recommended.

```python
class Box:
    def __init__(self, w, h, d):
        self._w, self._h, self._d = w, h, d

    def getVolume(self):
        return self._w * self._h * self._d

b = Box(10, 20, 30)
print(b._w, b._h, b._d)
```

Double underscore

- If we add one more underscore, the members will be hidden outside of the class and not accessible:

```
class Box:
    def __init__(self, w, h, d):
        self.__w, self.__h, self.__d = w, h, d

    def getVolume(self):
        return self.__w * self.__h * self.__d
```

- In this case, `__w`, `__h`, and `__d` can only be used internally by the class, the following code will fail:

```
b = Box(10, 20, 30)
print(b.__w, b.__h, b.__d)
```

Mangled

- In fact, the class members are not really hidden, Python only mangled their names so that we cannot use them directly.
- The `dir()` function shows all member of a class:

```
b = Box(10, 20, 30)
print(dir(b))
```

Output (partial only):

```
['_Box__d', '_Box__h', '_Box__w', ... 'getVolume']
```

- You can see that the three hidden attribtes are actually just renamed.

Double-double underscore

- Names in the form of `__XX__` are defined by Python as internal.
- `__init__()` is one of the example. These internal functions, although starting with double underscore, are not mangled.
- These functions control the behaviours of the structure.

Printing an object

- Another example of internal function is the `__str__()` function.
- Consider the previous `Box` class, if we print an object directly, the output does not look good:

```
b = Box(10, 20, 30)
print(b)
```

Output:

```
<__main__.Box object at 0x000001DF0B64CF28>
```

str

- We may implement the `__str__()` function to specify how an object can be presented.

```
class Box:
    # omitted
    def __str__(self):
        return f'A {self.__w} x {self.__h} x {self.__d} Box'

b = Box(10, 20, 30)
print(b)
```

Output:

```
A 10 x 20 x 30 Box
```

Exercises

Exercise 1 (1)

- Define a class `Frac` modelling a fraction with integer numerator and denominator. It supports the following:

Exercise 1 (2)

- `x = Frac(1, 2)` creates a fraction of $1/2$.
- `print(x)` prints fraction `x` in the form of a/b in the simplest form. For example:

```
x, y = Frac(2, 4), Frac(6, 3)
print(x, y)
```

Output:

```
1/2 2/1
```

Exercise 1 (3)

- `x.invert()` will replace fraction `x` by its reciprocal.

```
x = Frac(2, 3)
x.invert()
print(x)
```

Output:

```
3/2
```

Exercise 1 (4)

- `x.multi(y)` multiply fraction `y` to fraction `x`, fraction `x` will be updated.

```
x, y = Frac(1, 3), Frac(1, 2)
x.multi(y)
print(x)
```

Output:

```
1/6
```

Exercise 1 (5)

- `x.add(y)` add fraction `y` to fraction `x`, fraction `x` will be updated.

```
x, y = Frac(1, 3), Frac(1, 2)
x.add(x)
print(x)
x.add(y)
print(x)
```

Output:

```
2/3
7/6
```

Exercise 1 (6)

- `x.eval()` will return the decimal form of the fraction.

```
x = Frac(1, 8)
print(x.eval())
```

Output:

```
0.125
```

Exercise 1 test case

```python
n = 5
x = Frac(1, 1)
for i in range(2, n):
    x.add(Frac(1, i))
    x.multi(Frac(1, 2))
    x.invert()
print(n, x, x.eval())
```

Output:

```
5 40/29 1.3793103448275863
```

Advanced topics

- Remember: you may skip this section and come back later.

Static & class methods

Using the Class itself

- When we define a class, the class itself is actually a kind of object.
- It also owns all the members.

```
class Box:
    w, h, d = 10, 20, 5

print(Box.w, Box.h, Box.d)
```

Output:

```
10 20 5
```

No self

- If we use the class name to access a function defined in a class, in that case the function should not include the `self` parameter.

```
class Box:
    w, h, d = 10, 20, 5
    def explain():
        print('This is a Box class.')
        print(f'Default size: {Box.w} x {Box.h} x {Box.d}.')

Box.explain()
```

Method and self

- In fact, when a method is called from an object, it is the same as if we call the method through the class name, then provide the object as the first argument:

```
b = Box()
print(b.getVolume())
print(Box.getVolume(b))
```

- In an other words, `self` is automatically applied by Python when a method is called from an object.
- These methods can be called instance methods, as they work on instances of classes (i.e., objects).

Static methods

- The previously defined `explain()` method cannot be used by objects of the class because of the automatic `self` argument.

```
b = Box()
b.explain()  # this will fail
```

- To allow any objects to use it, we can add a decorator `@staticmethod` before the definion.

```
    @staticmethod
    def explain():
        print('This is a Box class.')
        print(f'Default size: {Box.w} x {Box.h} x {Box.d}.')
```

Class methods

- In a static method, we need to know the class name to access its members. That will be a problem if the class name changes.
- We can use the `@classmethod` decorator instead to make it a class method. A class method will add the class as the first argument automatically when executed.

```
    @classmethod
    def explain(cls):
        print('This is a Box class.')
        print(f'Default size: {cls.w} x {cls.h} x {cls.d}.')
```

Inheritance

Inheritance (1)

- We can create a class based on another class by adding the base class during class definition. For example to define class `Dog` that extends from class `Animal`:

```
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def bark(self):
        print(f'{self.name}: woof!')
```

Inheritance (2)

- In the previous example, `Dog` class will inherits the constructor, so we can create an instance of Dog like this:

```
d = Dog('Jimmy')
```

- And since `d` is an object of class `Dog`, it can use the methods defines in `Dog`.

```
d.bark()
```

- Note that `bark()` is using the attributes initialize in the constructor of `Animal`.

Method overriding

- When we extends a class, we can define a new method to override the original ones.
- Here is an example that overrides the constructor so that parameter `name` is optional:

```
class Dog(Animal):
    def __init__(self, name='Jo'):
        self.name = name
        self.sound = 'woof!'
```

super()

- We can use `super()` to access the base clasee.
- For example this will use the constructor of the base class to initialize name instead:

```
class Dog(Animal):
    def __init__(self, name='Jo'):
        super().__init__(name)
        self.sound = 'woof!'

    def bark(self):
        print(f'{self.name}: {self.sound}!')

d = Dog()
d.bark()
```

Duck typing

Duck typing

> *If it walks like a duck and it quacks like a duck, then it must be a duck*

- Python uses a principle called **Duck typing**.
- When we define functions that takes object as arguments, we don't need to care about the type of the arguments, we only care if the object provide the necessary features.

Duck typing example (1)

- Consider the following function that works on lists:

```
def countPositive(numbers):
    count = 0
    for n in numbers:
        if n > 0:
            count += 1
    return count
```

- Although the function is designed for list input, the code works perfectly when `numbers` can be iterated by a for-loop and all items are numerical.
- Therefore, the function work for any list, tuple, range, or even dictionary etc.

Duck typing example (2)

- Another example goes to the `print()` function.
- The `print()` function do not care what the type of the argument is, it will always use the result from `__str()__` of the value.

```python
class A:
    def __str__(self):
        return "A"

class B:
    def __str__(self):
        return "B"

a, b = A(), B()
print(a, b)
```

Errors and Exceptions

Errors and Exceptions

- During the course of the workshop, you should have already encountered a lot of errors.
- There are mainly two types of errors, **syntax error** and **runtime error**.

Syntax errors

- **Syntax errors** are errors in the syntax, codes cannot be executed if the code is incorrect in syntax:

```
a = 1
if a < 0
    print(a, 'is negative')
```

- Note the missing colon `:` for the if-statement.

Runtime error

- **Runtime errors** are errors happens when code is executed, for example when you divide a value by zero, trying access a variable that is not defined, etc.:

```
a = 1
print(a/0)
```

Exception handling

- We can wrap our code with `try...except` to capture errors and handle them:

```
try:
    a = int(input())
    print(1/a)
except:
    print('What have you done?')
```

- Try to input `0` or `a` for the code above. You will get the custom message instead of the error message by Python.

Exception handling

- You can capture specific errors instead:

```
try:
    a = int(input())
    print(1/a)
except ZeroDivisionError:
    print('I can't handle zero!')
except ValueError:
    print('I need a number!')
except:
    print('What have you done?')
```

- The last `except` capture all other errors that is not captured.

Exception handling

- You can capture multiple errors with the same `except`:

```
try:
    a = int(input())
    print(1/a)
except (ZeroDivisionError, ValueError):
    print('I need a number but no zero please!')
except:
    print('What have you done?')
```

Exception info

- We can assign a variable for the exception caught, in order to collect information from it.

```
try:
    a = int(input())
    print(1/a)
except (ZeroDivisionError, ValueError) as err:
    print('Error captured:', err)
    print('I need a number but no zero please!')
except:
    print('What have you done?')
```

Raise exception

- We can raise an exception ourselves using `raise`.

```python
try:
    a = int(input())
    if a == 1:
        raise ValueError("I don't like 1.")
except ValueError as err:
    print('Error captured:', err)
```

- Try inputing `1` and see the result.

Custom exception

- We can define our own error by extending the Exception class.

```python
class MyException(Exception):
    pass

try:
    a = int(input())
    if a == 1:
        raise MyException("I don't like 1.")
except MyException as err:
    print('Error captured:', err)
```

else

- We can add an `else` clause at the end, which will be executed when the `try` block finished successfully without catching an exception.

```
try:
    a = int(input())
    b = 1/a
except (ZeroDivisionError, ValueError):
    print('I need a number but no zero please!')
else:
    print('1 over', a, 'is', b)
```

- This is useful in presenting a result after `try-catch`

finally

- We can add an `finally` clause at the end, which will always be executed at the end.

```python
try:
    a = int(input())
    b = 1/a
except (ZeroDivisionError, ValueError):
    print('I need a number but no zero please!')
else:
    print('1 over', a, 'is', b)
finally:
    print('done.')
```

- This is useful for clean up purpose, for example if we opened a file or a network connection in `try`, we can clean them up in `finally`.

File IO

- File IO usually requires exception handling and therefore it is introduced here at the end.

Open a file

- To open a file, we can use the `open()` function. A file name and a mode should be specified.

```
f = open('test.txt', 'r')
f.close()
```

- `r` is the mode of accessing the file. This can be `r` for reading, `w` for writing, and `r+` for both.
- File must be closed with `close()` after accessing.

Using `with`

- We can also use a `with` block for file access, file will be automatically closed in this case:

```
with open('test.txt', 'r') as f:
    pass
```

- This is the preferred way of accessing a file.

File reading

- To read the entire file, we can use `read()`:

```python
with open('test.txt', 'r') as f:
    s = f.read()
    print(s)
```

- Alternatively we can read one line instead using `readline()`:

```python
with open('test.txt', 'r') as f:
    line = f.readline()
    print(line)
```

Iterating a file object

- In fact, the file object can be iterated. In that case, the file is read line by line.

```
with open('test.txt', 'r') as f:
    for line in f:
        print(line)
```

Writing to file

- The `write()` function write contents to a file.
- Note that opening a file in `w` mode will overwrite the whole file. You should use `r+` mode if you are only modifying a file.

```
with open('test.txt', 'w') as f:
    f.write("Hello, world")
```

Seeking

- The `seek()` function moves the current read/write position of a file.
- It takes 2 arguments, the first one is the movement, the second one is the starting point of movement, with `0` being the beginning of the file, `1` being the current position, and `2` being the end of file.
- For example we can append to a file like this:

```
with open('test.txt', 'r+') as f:
    f.seek(0, 2)
    f.write("Hello, world")
```