> Python workshop
>
> ## *3-3. Classes*

Introduction

Defining classes

We can define a class to model properties and bahaviours of a concept. For example, a **Box** has 3 dimensions and we can find the volume of it, so we can define a **Box** class in this way:

In [1]:
```
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
```

- `self` is a special keyword that is used in functions defined in a class.
- `__init__()` defines how a **Box** could be initialized. In this case, it needs 3 parameters, `width`, `height`, `depth`.
- `__init__()` initialize the internal property of the **Box** with the 3 specified values.
- `getVolume()` is a function that returns the volumne based on the corresponding properties in the class.

Using classes

With the **Box** class previously defined, we can create boxes of different sizes.

In [2]:
```
b1 = Box(3, 4, 5)
print(b1.getVolume())
b2 = Box(10, 20, 30)
print(b2.getVolume())
```

```
60
6000
```

Quick Quiz

What is the output of the following program?

```python
class MyClass:
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def getVal(self):
        return self.a * self.b

c = MyClass(1, 2)
d = MyClass(3, 4)
print(c.getVal() + d.getVal())
```

A	B	C	D
10	14	21	24

Demonstration 3-3

Define a class `Frac` modelling a fraction with integer numerator and denominator as explained below.

- `Frac(n, d)` creates a fraction that represents $\frac{n}{d}$.
- Define functions `n()` and `d()` that gives the numerator and denominator of the fraction. $\frac{n}{d}$ must be in its simplest form.
- Define function `invert()` that gives a new fraction that represents the recipical of the current fraction. For example, `Frac(2, 3).invert()` should returns a fraction that represents $\frac{3}{2}$.

Self-learning topics

Classes

- Class members
- Constructors
- String reprentation
- Modules

Class members

Attributes / properties

Attributes, or **properties** 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 `.`.

In [3]:
```
class Box:
    width = 10
    height = 20
    depth = 5

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

```
10 20 5
```

We can modify values of an attributes also.

In [4]:
```
class Box:
    width = 10
    height = 20
    depth = 5

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

```
10 20 50
```

Methods

Methods, or **member functions** define behaviours of an object. Results may be affected by the attributes of the object. 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 represents the object used to call the function. With `self`, the method can return results base on the values in the object.

Methods: example

In the example below, when `b1.getVolume()` is called, the `self` variable in `getVolume()` refers to `b1`. Thus `getVolume()` can return the volume according to the attribute values in `b1` and `b2` accordingly.

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

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

```
1000
10000
```

Constructor

Attribute creation in method

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

In [6]:
```
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())
```

```
6000
```

Customizing constructor (`__init__()`)

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.

Constructor: example

The previous example can then be modified to:

In [7]:
```
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())
```

```
6000
```

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:

In [8]:
```
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())
```

```
1 6000
```

String representation of an object (`__str()__`)

Consider the previous `Box` class, if we print an object directly, the output does not look good:

In [9]:
```
b = Box(10, 20, 30)
print(b)
```

```
<__main__.Box object at 0x0000018F551A73D0>
```

We may implement a special function `__str__()` to specify the string representation of an object. This allows objects to be printed directly.

In [10]:
```
class Box:
    def __init__(self, w, h, d):
        self.__w, self.__h, self.__d = w, h, d
    def __str__(self):
        return f'A box sized {self.__w} x {self.__h} x {self.__d}'

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

```
A box sized 10 x 20 x 30
```

Modules

It is possible to reuse functions or classes defined in another file. Suppose you have written the following function in a file named `helloUtil.py`.

```
def hello(hello='hello', name='David', message='How are you'):
    print(hello, name)
    print(message)
```

In another file, you can import this function by the statement:

```
from helloUtil import hello
hello()
```

Import *

The statement `from XX import YY` specify that we import the name `YY` from file `XX.py`.

- Name `YY` can be any variable or function name.
- File `XX.py` will be retrived in the same folder of the code, or the library path.
- We can import all names with `from XX import *`.

Note that `import *` will not import private/hidden functions that starts with an underscore `_`.

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()
```

Name clashes and `import ... as`

Consider this example:

```
def hello():
    print('Hello!')

from helloUtil import hello
hello()
```

The name `hello` is overwritten by import, and so the original definition is gone. To avoid this, we can use `as` to rename the imported function.

```
def hello():
    print('Hello!')

from helloUtil import hello as hello2
hello()
hello2()
```

Import module

Using `import *` is convenient but it will import all names from the file which may never be used, which is not desirable. One possible solution is to import the file as a **module** instead.

```
import helloUtil
helloUtil.hello()
```

Note that in this case, we need to access the functions from the module name instead.

Similarly, we can import a module and rename it using `as`.

```
import helloUtil as hello
hello.hello()
```

Self-evaluation exercises (3-3)

Quiz 3-3

Consider the following class that model a simple counter.

```
class Counter:
  tick = 0
  def click(self):
    # ...
```

1. Give a statement in place of `# ...` so that the counter tick is increased by 1. Remove all unnecessary space in your answer.
2. What is the name of the member function one should implement if object of this class could be printed directly to output its tick? For example, we hope the follwing code could be executed.

```
c = Counter()
c.click()
print(c)  # outputs 1
```

Exercise 3-3

Continue with the definition of the class `Frac` in Demo 3-3. Add the following features:

- Allow objects of `Frac` class to be printed directly. E.g., `print(Frac(1, 2))` shoud print `1/2`.
- Define function `averageWith()` that takes one object of `Frac` class and return the average of the current fraction with the input. E.g., `Frac(1, 4).averageWith(Frac(3, 4))` should gives an object of `Frac` class that represents $\frac{1}{2}$.

Using the `Frac` class defined, complete your program with the following main routine that calculates the harmonic mean of two input integers.

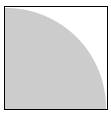
```
a = int(input())
b = int(input())

print(Frac(1, a).averageWith(1, b).invert())
```

Challenge 3-3

Monte Carlo method

A Monte Carlo method is to solve a problem by using random events. In this exercise we are going to find π using this method.

Consider a square of size 1×1 with a quater of a circle with raduis 1 as shown below.



By considering the area of the two regions in the figure, the chance of a random dot falling in the shaded region equals $\pi/4$. Therefore, if we generate X random dots in the square, and counted that Y of them being in the circle, we can estimate that π equals $4Y/X$.

Pseudo code

Here is the pseudo code of the Monte Carlo method.

```
count = 0
while number of random points < target number of points
    generate random value x in range [0, 1)
    generate random value y in range [0, 1)
    if x^2 + y^2 < 1
        increment count

pi = 4 * count / total number of random points
```

Task

For this exercise, you need to prepare two files. Say, `ex36MC.py` and `ex36main.py`.

- `ex36MC.py` implements a function (or class) that computes π using the Monte Carlo method explained above, using a given number `n`.
- `ex36main.py` implements a program that reads `n` from input and computes π using the function (or class) in `ex36MC.py`.

Your `ex36main.py` will probably be in this form:

```
# ... (import function "findPi()" from ex36MC.py)

n = input()
pi = findPi(n)
print(pi)
```

or if it is done using a class:

```
# ... (import class "FindPi" from ex36MC.py)

n = input()
pi = FindPi(n)
print(pi.find())
```

Optional topics

- Naming convention
- Inheritance
- Duck typing

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 many cases, this is used to show the intention of the program design. There are also some implication when the entities are imported to other programs, which is discussed in an optional topic below.

Underscore in a class

Attributes and methods starting with `_` has a meaning that they are private. They are not supposed 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.

```
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
```

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

In [11]:
```
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)
```

```
10 20 30
```

Double underscore

If we add one more underscore, the members will be hidden outside of the class and not accessible. In this example below, `__w`, `__h`, and `__d` can only be used internally by the class, so the code will fail to run.

In [12]:
```
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)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[12], line 9
      6         return self.__w * self.__h * self.__d
      8 b = Box(10, 20, 30)
----> 9 print(b.__w, b.__h, b.__d)

AttributeError: 'Box' object has no attribute '__w'
```

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 available in an object. You can see that the three hidden attribtes are actually just renamed to something else and are still available.

In [13]:
```
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(dir(b))
print(b._Box__w, b._Box__h, b._Box__d)
```

```
['_Box__d', '_Box__h', '_Box__w', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__
gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module_
_', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'getVolume']
10 20 30
```

Double-double underscore

Names in the form of `__XX__` are defined by Python as internal. They exists for specific purposes. These internal functions, although starting with double underscore, are not mangled. `__init__()` is one of the example.

Optional: 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.

In [14]:
```
class Box:
    w, h, d = 10, 20, 5

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

```
10 20 5
```

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:

In [15]:
```
class Box:
    w, h, d = 10, 20, 5
    def getVolume(self):
        return self.w * self.h * self.d

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

```
1000
1000
```

In another 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).

No self

On the other hand, if a function is supposed to be called by the class name only, the function should not include the `self` parameter.

In [16]:
```
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()
```

```
This is a Box class.
Default size: 10 x 20 x 5.
```

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. The function becomes a static method.

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

In [17]:
```
class Box:
    w, h, d = 10, 20, 5
    @staticmethod
    def explain():
        print('This is a Box class.')
        print(f'Default size: {Box.w} x {Box.h} x {Box.d}.')

b = Box()
b.explain()
```

```
This is a Box class.
Default size: 10 x 20 x 5.
```

Class methods

Notice that in the example above, the function `explain()` needs to know the class name `Box` to access the three variables. This 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, similar to how `self` is automatically populated.

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

In [18]:
```
class Box:
    w, h, d = 10, 20, 5
    @classmethod
    def explain(c):
        print('This is a Box class.')
        print(f'Default size: {c.w} x {c.h} x {c.d}.')

Box.explain()
```

```
This is a Box class.
Default size: 10 x 20 x 5.
```

In [19]:
```
b = Box()
b.explain()
```

```
This is a Box class.
Default size: 10 x 20 x 5.
```

Optional: Inheritance

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!')
```

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`.

In [20]:
```
class Animal:
    def __init__(self, name):
        self.name = name
```

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

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

```
Jimmy: woof!
```

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:

In [21]:
```
class Animal:
    sound = "..."
    def __init__(self, name):
        self.name = name
    def bark(self):
        print(f'{self.name}: {self.sound}')

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

```
Jo: woof!
```

super()

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

In [22]:
```
class Animal:
    sound = "..."
    def __init__(self, name):
        self.name = name
    def bark(self):
        print(f'{self.name}: {self.sound}')

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

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

```
Jo: woof!
```

Optional: 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 some objects as arguments, we don't care about the type of the arguments, but instead, we care if the object provide the necessary features.

Duck typing example

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.

In [23]:
```
def countPositive(numbers):
    count = 0
    for n in numbers:
        if n > 0:
            count += 1

    return count

print(countPositive([-1, 1, -2, 2])) # list
print(countPositive((-1, 1, -2, 2))) # tuple
print(countPositive(range(-10, 10, 3))) # range
print(countPositive({1: 'a', -1: 'b'})) # dictionary
```

```
2
2
3
1
```

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.

In [24]:
```
class A:
    def __str__(self):
        return "A"

class B:
```

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

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

```
A B
```