1.2 Working with class and instance data – instance variables

Instance variables

This kind of variable exists when and only when it is explicitly created and added to an object. This can be done during the object's initialization, performed by the init method, or later at any moment of the object's life. Furthermore, any existing property can be removed at any time.

Each object carries its own set of variables – they don't interfere with one another in any way. The word instance suggests that they are closely connected to the objects (which are class instances), not to the classes themselves. To get access to the instance variable, you should address the variable in the following way: objectdotvariable_name.

Let's look at how the instance variable is created and accessed in the code presented below.

```
class Demo:
    def __init__(self, value):
        self.instance_var = value

d1 = Demo(100)
d2 = Demo(200)

print("d1's instance variable is equal to:", d1.instance_var)
print("d2's instance variable is equal to:", d2.instance_var)
```

1. **__init__** creates an instance_var variable for the instance. The keyword self is used to indicate that this variable is created coherently and individually for the instance to make it independent from other instances of the same class;
2. we instantiate the class twice, each time passing a different value to be stored inside the object;
3. the print instructions prove the fact that instance variable values are kept independently, because the printed values differ.

Another snippet shows that instance variables can be created during any moment of an object's life. Moreover, it lists the contents of each object, using the built-in dict property that is present for every Python object.

```
class Demo:
    def __init__(self, value):
        self.instance_var = value

d1 = Demo(100)
d2 = Demo(200)

d1.another_var = 'another variable in the object'

print('contents of d1:', d1.__dict__)
print('contents of d2:', d2.__dict__)
```

This example shows that modifying the instance variable of any object has no impact on all the remaining objects. Instance variables are completely isolated from each other.

output

```
contents of d1: {'instance_var': 100, 'another_var': 'another variable in the
object'}
contents of d2: {'instance_var': 200}
```

Class variables

Class variables are defined within the class construction, so these variables are available before any class instance is created. To get access to a class variable, simply access it using the class name, and then provide the variable name.

```
class Demo:
    class_var = 'shared variable'

print(Demo.class_var)
print(Demo.__dict__)
```

Similarly to instance variables, class variables are shown in the class's **__dict__** dictionary.

output

```
shared variable
{'__module__': '__main__', 'class_var': 'shared variable', '__dict__': ,
'__weakref__': , '__doc__': None}
```

As a class variable is present before any instance of the class is created, it can be used to store some meta data relevant to the class, rather than to the instances:

- fixed information like description, configuration, or identification values;
- mutable information like the number of instances created (if we add a code to increment the value of a designated variable every time we create a class instance)

A class variable is a class property that exists in just one copy, and it is stored outside any class instance. Because it is owned by the class itself, all class variables are shared by all instances of the class. They will therefore generally have the same value for every instance; butas the class variable is defined outside the object, it is not listed in the object's __dict__.

```
class Demo:
    class_var = 'shared variable'

d1 = Demo()
d2 = Demo()

print(Demo.class_var)
print(d1.class_var)
print(d2.class_var)

print('contents of d1:', d1.__dict__)
```

output

```
shared variable
```

```
shared variable
shared variable
contents of d1: {}
```

Conclusion: when you want to read the class variable value, you can use a class or class instance to access it.

When you want to set or change a value of the class variable, you should access it via the class, but not the class instance, as you can do for reading.

When you try to set a value for the class variable using the object (a variable referring to the object or `self` keyword) but not the class, you are creating an instance variable that holds the same name as the class variable. The following snippet shows such a case – remember this in order to avoid wasting time hunting for bugs!

The snippet contains additional comments for direct explanations.

```
class Demo:
    class_var = 'shared variable'

d1 = Demo()
d2 = Demo()

# both instances allow access to the class variable
print(d1.class_var)
print(d2.class_var)
print('.' * 20)

# d1 object has no instance variable
print('contents of d1:', d1.__dict__)
print('.' * 20)

# d1 object receives an instance variable named 'class_var'
d1.class_var = "I'm messing with the class variable"

# d1 object owns the variable named 'class_var' which holds a different value than
the class variable named in the same way
print('contents of d1:', d1.__dict__)
print(d1.class_var)
print('.' * 20)

# d2 object variables were not influenced
print('contents of d2:', d2.__dict__)

# d2 object variables were not influenced
print('contents of class variable accessed via d2:', d2.class_var)
```

output

```
shared variable
shared variable
....................
contents of d1: {}
....................
contents of d1: {'class_var': "I'm messing with the class variable"}
I'm messing with the class variable
```

```
.....................
contents of d2: {}
contents of class variable accessed via d2: shared variable
```

Class variables and instance variables are often utilized at the same time, but for different purposes. As mentioned before, class variables can refer to some meta information or common information shared amongst instances of the same class.

The example below demonstrates both topics: each class owns a counter variable that holds the number of class instances created. Moreover, each class owns information that helps identify the class instance origins. Similar functionality could be achieved with the `isinstance()` function, but we want to check if class variables can be helpful in this domain.

Both the **Duck** and **Chicken** classes own a class variable named **species**, which holds a value unique to each class. When we iterate over all objects, we can examine the value of this variable to take appropriate action.

```
class Duck:
    counter = 0
    species = 'duck'

    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex
        Duck.counter +=1

    def walk(self):
        pass

    def quack(self):
        print('quacks')

class Chicken:
    species = 'chicken'

    def walk(self):
        pass

    def cluck(self):
        print('clucks')

duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")

chicken = Chicken()

print('So many ducks were born:', Duck.counter)

for poultry in duckling, drake, hen, chicken:
    print(poultry.species, end=' ')
    if poultry.species == 'duck':
        poultry.quack()
    elif poultry.species == 'chicken':
        poultry.cluck()
```

output

```
So many ducks were born: 3
duck quacks
duck quacks
duck quacks
chicken clucks
```

Another example shows that a class variable of a super class can be used to count the number of all objects created from the descendant classes (subclasses). We'll achieve this by calling the superclass __init__ method.

Another class variable is used to keep track of the serial numbers (which in fact are also counters) of particular subclass instances. In this example, we are also storing instance data (phone numbers) in instance variables.

The class **Phone** is a class representing a blueprint of generic devices used for calling. This class definition delivers the **call** method, which displays the object's variable, which holds the phone number. This class also holds a class variable that is used to count the number of instances created by its subclasses.

Subclasses make use of the superclass __init__ method, then instances are created. This gives us the possibility to increment the superclass variable.

```
class Phone:
    counter = 0

    def __init__(self, number):
        self.number = number
        Phone.counter += 1

    def call(self, number):
        message = 'Calling {} using own number {}'.format(number, self.number)
        return message


class FixedPhone(Phone):
    last_SN = 0

    def __init__(self, number):
        super().__init__(number)
        FixedPhone.last_SN += 1
        self.SN = 'FP-{}'.format(FixedPhone.last_SN)


class MobilePhone(Phone):
    last_SN = 0

    def __init__(self, number):
        super().__init__(number)
        MobilePhone.last_SN += 1
        self.SN = 'MP-{}'.format(MobilePhone.last_SN)


print('Total number of phone devices created:', Phone.counter)
print('Creating 2 devices')
fphone = FixedPhone('555-2368')
mphone = MobilePhone('01632-960004')
```

```
print('Total number of phone devices created:', Phone.counter)
print('Total number of mobile phones created:', MobilePhone.last_SN)

print(fphone.call('01632-960004'))
print('Fixed phone received "{}" serial number'.format(fphone.SN))
print('Mobile phone received "{}" serial number'.format(mphone.SN))
```

output

```
Total number of phone devices created: 0
Creating 2 devices
Total number of phone devices created: 2
Total number of mobile phones created: 1
Calling 01632-960004 using own number 555-2368
Fixed phone received "FP-1" serial number
Mobile phone received "MP-1" serial number
```

LAB

Scenario Imagine that you receive a task description of an application that monitors the process of apple packaging before the apples are sent to a shop.

A shop owner has asked for 1000 apples, but the total weight limitation cannot exceed 300 units.

Write a code that creates objects representing apples as long as both limitations are met. When any limitation is exceeded, than the packaging process is stopped, and your application should print the number of apple class objects created, and the total weight.

Your application should keep track of two parameters:

- the number of apples processed, stored as a class variable;
- the total weight of the apples processed; stored as a class variable. Assume that each apple's weight is random, and can vary between 0.2 and 0.5 of an imaginary weight unit;

Hint: Use a `random.uniform(lower, upper)` function to create a random number between the lower and upper float values.

```
import random

class Pomes:
    total_w = 0
    total_i = 0

    def __init__(self):
        if (Pomes.total_w > 300):
            print("stop:%s %s" % (Pomes.total_w,Pomes.total_i) )
        else:
            self.weight = random.uniform(0.2,0.5)
            Pomes.total_w += self.weight
            Pomes.total_i += 1



for i in range(1000):
```

```
    Pomes()
```