Chapter 19
Class Side and Static Behaviour

19.1 Introduction

Python classes can hold data and behaviour that is not part of an instance or object; instead they are part of the class.

This chapter introduces class side data, behaviour and static behaviour.

19.2 Class Side Data

In Python classes can also have attributes; these are referred to as *class* variables or attributes (as opposed to instance variables or attributes).

In Python variables defined within the scope of the class, but outside of any methods, are tied to the class rather than to any instance and are thus class variables.

For example, we can update the class `Person` to keep a count of how many instances of the class are created:

```
class Person:
    """ An example class to hold a persons name and age"""

    instance_count = 0

    def __init__(self, name, age):
        Person.instance_count += 1
        self.name = name
        self.age = age
```

The variable `instance_count` is not part of an individual object, rather it is part of the class and all instances of the class can access that shared variable by prefixing it with the class name.

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_19

Now each time a new instance of the class is created, the `instance_count` is incremented, thus if we write:

```
p1 = Person('Jason', 36)
p2 = Person('Carol', 21)
p3 = Person('James', 19)
p4 = Person('Tom', 31)
print(Person.instance_count)
```

The output will be:

```
4
```

This is because 4 instances have been created and thus `__init__()` has been run 4 times and `instance_count` has been incremented four times.

19.3 Class Side Methods

It is also possible to define behaviour that is linked to the *class* rather than an individual object; this behaviour is defined in a class method.

Class methods are written in a similar manner to any other method but are *decorated* with `@classmethod` and take a first parameter which represents the class rather than an individual instance. This decoration is written before the method declaration.

An example of a class method is shown below:

```
class Person:
    """ An example class to hold a persons name and age"""
    instance_count = 0

    @classmethod
    def increment_instance_count(cls):
        cls.instance_count += 1

    def __init__(self, name, age):
        Person.increment_instance_count()
        self.name = name
        self.age = age
```

In this case the class method increments the `instance_count` variable; note that the `instance_count` variable is accessed via the `cls` parameter passed into the `increment_instance_count` method by Python. As this is a class method you do not need to prefix the class attribute with the name of class; instead the first parameter to the class method, `cls`, represents the class itself.

The class method can be accessed by *prefixing* it with the name of the class and using the *dot* notation to indicate which method to call. This is illustrated in the body of the `__init__()` method.

19.3.1 Why Class-Side Methods?

It may at first seem unclear what should normally go in an instance method as opposed to what should go in a class method. After all, they are both defined in the class. However, it is important to remember that

- Instance methods define the behaviour of the instance or object.
- Class methods define the behaviour of the class.

Class-side methods should only perform one of the following roles:

- *Instance creation* This role is very important as it is how you can use a class as a factory for objects and can help hide a whole load of set up and instantiation work.
- *Answering enquiries about the class* This role can provide generally useful objects, frequently derived from class variables. For example, they may return the number of instances of this class that have been created.
- *Instance management* In this role, class-side methods control the number of instances created. For example, a class may only allow a single instance of the class to be created; this is termed a singleton class. Instance management methods may also be used to access an instance (e.g. randomly or in a given state).
- *Examples* Occasionally, class methods are used to provide helpful examples which explain the operation of a class. This can be very good practice.
- *Testing* Class-side methods can be used to support the testing of an instance of a class. You can use them to create an instance, perform an operation and compare the result with a known value. If the values are different, the method can report an error. This is a very useful way of providing regression tests.
- *Support* for one of the above roles.

Any other tasks should be performed by an instance method.

19.4 Static Methods

There is one more type of method that can be defined on a class; these are *static* methods.

Static methods are defined within a class but are not tied to either the class nor any instance of the class; they do not receive the special first parameter representing either the class (cls for class methods) or the instances (self for instance methods).

They are in effect, the same as *free standing functions* but are defined within a class often for convenience or to provide a way to group such functions together.

A *static* method is a method that is decorated with the `@staticmethod` decorator.

An example of a static method is given below:

```
class Person:

    @staticmethod
    def static_function():
        print('Static method')
```

Static methods are invoked via the name of the class they are defined in, for example:

```
Person.static_function()
```

A note for Java and C# programmers; in both Java and C# the term class side and static are used interchangeably (not helped by the use of the keyword static for these methods). However, in both cases those methods are the equivalent of class side methods in Python. In Python *class* methods and *static* methods are two very, very different things—do not use these terms interchangeably.

19.5 Hints

There are a range of special methods available in Python on a class.

All of these special methods start and end with a double underbars (`'__'`).

In general, in Python anything that starts and ends with these double underbars is considered special and so care should be taken when using them.

You should never name one of your own methods or functions `__<something>__` unless you intend to (re)define some default behaviour.

19.6 Online Resources

There is surprisingly little information available on Python's static and class methods (which in part explains why many developers are confused by them), however the following are available:

- https://python-reference.readthedocs.io/en/latest/docs/functions/staticmethod.html documentation on static methods.
- https://python-reference.readthedocs.io/en/latest/docs/functions/classmethod.html?highlight=classmethod documentation on class methods.
- https://www.tutorialspoint.com/class-method-vs-static-method-in-python tutorial on class methods versus static methods.

19.7 Exercises

The aim of this exercise is to add housekeeping style methods to the `Account` class. You should follow these steps:

1. We want to allow the `Account` class from the last chapter to keep track of the number of instances of the class that have been created.
2. Print out a message each time a new instance of the `Account` class is created.
3. Print out the number of accounts created at the end of the previous test program. For example add the following two statements to the end of the program:

```
print('Number of Account instances created:',
Account.instance_count)
```