Chapter 26
Abstract Base Classes

26.1 Introduction

This chapter presents *Abstract Base Classes* (also known as ABCs) which were originally introduced in Python 2.6.

An Abstract Base Class is a class that you cannot instantiate and that is expected to be extended by one or more subclassed. These subclasses will then fill in any the gaps left the base class.

Abstract Base Classes are very useful for creating class hierarchies with a high level of reuse from the root class in the hierarchy.

26.2 Abstract Classes as a Concept

An abstract class is a class from which you cannot create an object. It is typically missing one or more elements required to create a fully functioning object.

In contrast a non-abstract (or concrete) class leaves nothing undefined and can be used to create a working object.

You may therefore wonder what use an abstract class is?

The answer is that you can group together elements which are to be shared amongst a number of classes, without providing a complete implementation. In addition, you can force subclasses to provide specific methods ensuring that implementers of a subclass at least supply appropriately named methods. You should therefore use abstract classes when:

- you wish to specify data or behaviour common to a set of classes, but insufficient for a single instance,
- you wish to force subclasses to provide specific behaviour.

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_26

In many cases, the two situations go together. Typically, the aspects of the class to be defined as abstract are specific to each class, while what has been implemented is common to all classes.

26.3 Abstract Base Classes in Python

Abstract Base Classes (or ABCs as they are sometimes referred to) cannot be instantiated themselves but can be extended by subclasses. These subclasses can be concrete classes or can themselves be Abstract Base Classes (that extend the concept defined in the root Abstract Base Class).

Abstract Base Classes can be used to define generic (potentially abstract) behaviour that can be mixed into other Python classes and act as an abstract root of a class hierarchy. They can also be used to provide a more formal way of specifying behaviour that must be provided by a concrete class.

Abstract Base Classes can have:

- Zero or more abstract methods or properties (but are not required to).
- Zero or more concrete methods and properties (but are not required to).
- Both *private* and *protected* attributes (following the single underscore and double underscore conventions).

ABCs can also be used to specify a specific interface or formal protocol. If an ABC defines any abstract methods or abstract properties, then the subclasses must provide implementations for all such abstract elements.

There are many built-in ABCs in Python including (but not limited to):

- data structures (`collection` module),
- `numbers` module,
- streams (`IO` module).

In fact, ABCs are widely used internally within Python itself and many developers use ABCs without ever knowing they exist or understanding how to define them.

Indeed, ABCs are not widely used by developers building systems with Python although this is in part because they are most appropriate to those building libraries, particularly those that are expected to be extended by developers themselves.

26.3.1 Subclassing an ABC

Typically, an Abstract Base Class will need to be imported from the module in which it is defined; of course, if the ABC is defined in the current module then this will not be necessary.

As an example, the `collections.MutableSequence` class is an ABC; this is an ABC for a sequence of elements that can be modified (mutable) and iterated over. We can use this as the base class for our own type of collection which we will call a `Bag`, for example:

```
from collections import MutableSequence

class Bag(MutableSequence):
    pass
```

In this example we are importing the `MutableSequence` from the `collections` module. We then define the class `Bag` as extending the `MutableSequence` Abstract Base Class. For the moment we are using the special Python keyword `pass` as a place holder for the body of the class.

However, this means that the `Bag` class is actually also an abstract class as it does not implement any of the abstract methods in the `MutableSequence` ABC.

Python, however, does not validate this at *import* time; instead it validates it at *runtime* when an instance of the type is to be created.

In this case, the `Bag` class does not implement the abstract methods in `MutableSequence` and thus if a program attempts to create an instance of `Bag`, then the following error would be raised:

```
Traceback (most recent call last):
  File "/pythonintro/abstract/Bag.py", line 10, in <module>
    main()
  File "/pythonintro/abstract/Bag.py", line 7, in main
    bag = Bag()
TypeError: Can't instantiate abstract class Bag with abstract
methods __delitem__, __getitem__, __len__, __setitem__,
insert
```

As can be seen this is a rather *formal* requirement; if you don't implement all the methods defined as abstract in the parent class then you can't create an instance of the class you are defining (because it is also abstract).

We can define a method for each of the abstract classes in the `Bag` class and then we will be able to create an instance of the class, for example:

```
from collections import MutableSequence

class Bag(MutableSequence):

    def __getitem__(self, index):
        pass

    def __delitem__(self, index):
        pass

    def __len__(self):
        pass

    def __setitem__(self, index, value):
        pass

    def insert(self, index, value):
        pass
```

This version of `Bag` meets all the requirements imposed on it by the ABC `MutableSequence`; that is, it implements each of the special methods listed and the `insert` method. The class `Bag` can now be considered to be a concrete class.

However, in this case the methods themselves don't do anything (they again use the Python keyword `pass` which acts as a placeholder for the code to implement each method). However, we can now write:

```
bag = Bag()
```

And the application will not generate an error message.

At this point we could now progress to implementing each method such that it provides an appropriate implementation of the `Bag`.

26.3.2 Defining an Abstract Base Class

An Abstract Base Class can be defined by specifying that the class has a metaclass; typically, `ABCMeta`. The `ABCMeta` metaclass is provided by the `abc` module.

The metaclass is specified using the metaclass attribute of the parent class list. This will create a class that can be used as an ABC. Alternatively you can extend the abc.ABC class that specified the ABCMeta as its metaclass.

This is exactly how the ABCs in the `_collections_abc.py` file are implemented.

The following code snippet illustrates this idea. The `ABCMeta` class is imported from the `abc` module. It is then used with the class `Shape` via the `metaclass` attribute of the class inheritance list:

```
from abc import ABCMeta

class Shape(metaclass=ABCMeta):

    def __init__(self, id):
        self.id = id
```

Note that at this point although `Shape` is an ABC, it does not define any abstract elements and so can actually be instantiated just like any other concrete class. However, we will next define some abstract methods for the `Shape` ABC.

To define an abstract method, we need to also import the `abstractmethod` decorator from the `abc` module, (if we want to define an abstract property then we need to add `@property` to an appropriate abstract method). Importing the `abstractmethod` decorator is illustrated below:

```
from abc import ABCMeta, abstractmethod
```

We can now expand the definition of our `Shape` class:

```
from abc import ABCMeta, abstractmethod

class Shape(metaclass=ABCMeta):
    def __init__(self, id):
        self._id = id

    @abstractmethod
    def display(self): pass

    @property
    @abstractmethod
    def id(self): pass
```

The class `Shape` is now an Abstract Base Class and requires that any subclass must provide an implementation of the method `display()` and the property `id` (otherwise the subclass will automatically become abstract).

The class `Circle` is a concrete subclass of the `Shape` ABC; it thus provides a `__init__()` initialisation method, a `display` method and an `id` property (the `__init__()` method is used to allow the `_id` attribute in the base class to be initialised).

```
class Circle(Shape):
    def __init__(self, id):
        super().__init__(id)

    def display(self):
        print('Circle: ', self._id)

    @property
    def id(self):
        """ the id property """
        return self._id
```

We can now use the class `Circle` in an application:

```
c = Circle("circle1")
print(c.id)
c.display()
```

We can instantiate the `Circle` class as it is concrete and we know we can call the method `display()` and access the property `id` on instances of `Circle`. The output from the above code is thus:

```
circle1
Circle:  circle1
```

26.4 Defining an Interface

Many languages such as Java and C# have the concept of an *interface* definition; this is a contract between the implementors of an interface and the user of the implementation guaranteeing that certain facilities will be provided.

Python does *not* explicitly have the concept of an interface contract (note here interface refers to the interface between a class and the code that utilizes that class).

However, it does have Abstract Base Classes.

Any Abstract Base Class that only has abstract methods or properties in it can be treated as a contract that must be implemented (it can of course also have concrete methods, properties and attributes; it is up to the developer). However, as we know that Python will guarantee that any instances can only be created from concrete classes, we can treat an ABC has behaving like a contract between a class and those using that class.

This is an approach that is adopted by numerous frameworks and libraries within Python.

26.5 Virtual Subclasses

In most object-oriented programming languages, for one class to be *treated* as a subclass of another class it is necessary for the subclass to *extend* the parent class. However, Python has a more relaxed approach to typing, as is illustrated by the idea of *Duck Typing* (discussed in the next chapter).

In some situations however, it is useful to be able to confirm that one type is a subclass of another or that an instance is an instance of a specific type (which may come from the object's class hierarchy) at runtime.

Indeed, in Python, it is not necessary to be an actual subclass of a parent class to be considered a subclass—instead *Virtual* subclasses allow one class to be treated as a subclass of another even though there is no direct inheritance relationship between them. The key here is that the *virtual* subclass must match the required *interface* presented by the virtual parent class.

This is done by registering one class as a Virtual Subclass of an Abstract Base Class. That is, the virtual parent class must be an ABC and then the subclass can be registered (at runtime) as a *virtual* subclass of the ABC. This is done using a method called `register()`.

Once a class is registered as a subclass of an ABC the `issubclass()` and `isintance()` methods will return `True` for that class with respect to the virtual parent class.

For example, given the following two currently independent classes:

```
from abc import ABCMeta

class Person(metaclass=ABCMeta):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def birthday(self):
        print('Happy Birthday')

class Employee(object):
    def __init__(self, name, age, id):
        self.name = name
        self.age = age
        self.id = id

    def birthday(self):
        print('Its your birthday')
```

If we now check to see if `Employee` is a subclass of `Person` we will get the value `False` returned. We will of course also get `False` if we check to see an instance of `Employee` is actually an instance of the class `Person`:

```
print(issubclass(Employee, Person))
e = Employee('Megan', 21, 'MS123')
print(isinstance(e, Person))
```

This will generate the output:

```
False
False
```

However, if we now register the class `Employee` as a virtual subclass of the class `Person`, the two test methods will return `True`:

```
Person.register(Employee)
print(issubclass(Employee, Person))
e = Employee('Megan', 21, 'MS123')
print(isinstance(e, Person))
```

Which now generates the output:

```
True
True
```

This provides a very useful level of flexibility that can be exploited when using existing libraries and frameworks.

26.6 Mixins

A *mixin* is a class that represents some (typically concrete) functionality that has the potentially to be useful in multiple situations but on its own is not something that would be instantiated.

However, a *mixin* can be mixed into other classes and can extend the data and behavior of that type and can access data and methods provided by those classes.

Mixins are a common category of Abstract Base Classes; although they are implicit in their use (and naming) rather than being a concrete construct within the Python language itself.

For example, let us define a `PrinterMixing` class that provides a utility method to be used with other classes. It is not something that we want developers to instantiate itself so we will make it an ABC but it does not define any abstract methods or properties (so there is nothing for a subclass to have to implement).

```
from abc import ABCMeta

class PrinterMixin(metaclass=ABCMeta):
    def print_me(self):
        print(self)
```

We can now use this with a class `Employee` that extends the class `Person` and mixes in the `PrinterMixin` class:

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

class Employee(Person, PrinterMixin):
    def __init__(self, name, age, id):
        super().__init__(name)
        self.age = age
        self.id = id

    def __str__(self):
        return 'Employee(' + self.id + ')' + self.name + '['
+ str(self.age) + ']'
```

This now means that when we instantiate the class `Employee` we can call the `print_me()` method on the `Employee` object:

```
e = Employee('Megan', 21, 'MS123')
e.print_me()
```

which will print out

```
Employee(MS123)Megan[21]
```

One point to note about the `PrinterMixin` is that it is completely independent of the class it is mixed into. However, mixins can also impose some *constraints* on the classes they will be mixed into. For example, the `IDPrinterMixin` shown below assumes that the class it will be mixed into has an attribute or property called `id`.

```
class IDPrinterMixin(metaclass=ABCMeta):
    def print_id(self):
        print(self.id)
```

This means that it cannot be mixed successfully into the class `Person`—if it was then when the `print_id()` method was called an error would be generated.

However, the class `Employee` does have an `id` attribute and thus the `IDPrinterMixin` can be mixed into the `Employee` class:

```
class Employee(Person, PrinterMixin, IDPrinterMixin):

    def __init__(self, name, age, id):
        super().__init__(name)
        self.age = age
        self.id = id

    def __str__(self):
        return 'Employee(' + self.id + ')' + self.name + '['
+ str(self.age) + ']'
```

Which means that we can now call write:

```
e = Employee('Megan', 21, 'MS123')
e.print_me()
e.print_id()
```

Which will generate:

```
Employee(MS123)Megan[21]
MS123
```

26.7 Online Resources

Some online references for Abstract Base Classes include:

- https://docs.python.org/3/library/abc.html The standard Library documentation on Abstract Base Classes.
- https://pymotw.com/3/abc/index.html The Python Module of the Week page for Abstract Base Classes.
- https://www.python.org/dev/peps/pep-3119/ Python PEP 3119 that introduced Abstract Base Classes.

26.8 Exercises

The aim of this exercise is to use an Abstract Base Class.

The `Account` class of the project you have been working on throughout the last few chapters is currently a concrete class and is indeed instantiated in our test application.

Modify the `Account` class so that it is an Abstract Base Class which will force all *concrete* examples to be a subclass of `Account`.

The account creation code element might now look like:

```
acc1 = accounts.CurrentAccount('123', 'John', 10.05, 100.0)
acc2 = accounts.DepositAccount('345', 'John', 23.55, 0.5)
acc3 = accounts.InvestmentAccount('567', 'Phoebe', 12.45,
'risky')
```