Chapter 27
Protocols, Polymorphism and Descriptors

27.1 Introduction

In this chapter we will explore the idea of an implicit contract between an object and the code that uses that object. As part of this discussion we will explore what is meant by Duck Typing. Following this we will introduce the Python concept called a *protocol*. We will explore its role within Python programming and look at two commonly occurring protocols; the *Context Manager Protocol* and the *Descriptor Protocol*.

27.2 Implicit Contracts

Some programming languages (most notable Java and C#) have the idea of an explicit contract between a class and the user of that class; this contract provides a guarantee of the methods that will be provided and the types that will be used for parameters and return values from these methods. In these languages it helps to guarantee that a method is only called with the appropriate type of values and only in appropriate situations. Slightly confusingly these contracts are referred to as *interfaces* in Java and C#; but are intended to describe the application programming *interface* presented by the class.

Python is a much more flexible and free flowing language than either Java or C# and so has no explicit concept of an *interface*. This can however make things more complex at times; for example, consider the very simple `Calculator` class given below:

```
class Calculator:
    def add(self, x, y):
        return x + y
```

© Springer Nature Switzerland AG 2019
J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_27

What are the valid values that can be passed into the add method and used for the parameters x and y?

Initially it might seem that numeric values such as 1, 2 and 3.4, 5.77 etc. would be the only things that can be used with the add method:

```
calc = Calculator()

print('calc.add(3, 4):', calc.add(3, 4))
print('calc.add(3, 4.5):', calc.add(3, 4.5))
print('calc.add(4.5, 6.2):', calc.add(4.5, 6.2))
print('calc.add(2.3, 7):', calc.add(2.3, 7))
print('calc.add(-1, 4):', calc.add(-1, 4))
```

This generates the following output:

```
calc.add(3, 4): 7
calc.add(3, 4.5): 7.5
calc.add(4.5, 6.2): 10.7
calc.add(2.3, 7): 9.3
calc.add(-1, 4): 3
```

However, this actually represents a contract that the values passed into the Calculator.add() method will support the *plus* operator. In an earlier chapter we explored a class Quantity which implemented this operator (among others) and thus we can also use Quantity objects with the Calculator's add() method:

```
q1 = Quantity(5)
q2 = Quantity(10)
print(calc.add(q1, q2))
```

Which prints out:

```
Quantity[15]
```

This implied contract says that the Calculator.add() method will work with anything that supports the numeric add operator (or to put it another); anything that is numeric like.

This is also known as *Duck Typing*; this is described in the next section.

27.3 Duck Typing

This rather strange term comes from an old saying:

If it walks like a duck, swims like a duck and quacks like a duck then it's a Duck!

In Python *Duck Typing* (also known as shape typing or structural typing) implies that if a object can perform the required set of operations then it's a suitable thing to use for what you want. For example, if your type can be used with the add, multiply, divide and subtract operators than it can be treated as a Numeric type (even if it isn't).

This is a very powerful feature of Python and allows code originally written to work with a specific set of types, to also be used with a completely new set of types; as long as they meet the *implicit* contract defined within the code.

It is also interesting to note, that a particular set of methods may have a *super set* of requirements on a particular type, but you only need to implement as much as is required for the functionality you will actually use.

For example, let us modify the `Calculator` class a bit and add some more methods to it:

```
class Calculator:
    """ Simple Calculator class"""
    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        return x - y

    def multiply(self, x, y):
        return x * y

    def divide(self, x, y):
        return x / y
```

At first sight this may indicate that anything being used with the `Calculator` must implement all four operators '+', '−', '/' and '*'. However, this is only true if you need to execute all four of the methods defined in the class.

For example, consider the type `Distance`:

```
class Distance:
    def __init__(self, d):
        self.value = d
    def __add__(self, other):
        return Distance(self.value + other.value)
    def __sub__(self, other):
        return Distance(self.value - other.value)
    def __str__(self):
        return 'Distance[' + str(self.value) + ']'
```

This defines a class that implements only the `__add__()` and `__sub__()` methods and will thus only support the '+' and '−' operators.

Can instances of `Distance` be used with the `Calculator` class? The answer is that they can but only with the `add` and `subtract` methods (as they only meet part of the implied contract between the `Calculator` class and any types used with that class).

We can thus write:

```
d1 = Distance(6)
d2 = Distance(3)
print(calc.add(d1, d2))
print(calc.subtract(d1, d2))
```

And obtain the output:

```
Distance[9]
Distance[3]
```

However, if we try to use the `multiply()` or `divide()` methods, we will get an error, for example:

```
Traceback (most recent call last):
  File "Calculator.py", line 46, in <module>
    print(calc.divide(d1, d2))
  File "Calculator.py", line 15, in divide
    return x / y
TypeError: unsupported operand type(s) for /:'Distance' and
'Distance'
```

Basically, it is telling you that the operator '/' is not supported when used with the `Distance` type.

27.4 Protocols

As mentioned above, Python does not have any formal mechanism for stating what is required between the supplier of some functionality and the user or consumer of that functionality. Instead the far less formal approach termed *Duck Typing* is adopted instead.

This however, raises the question; how do you know what is required? How do you know that you must provide the numeric operators for an object to be used with the `Calculator` class?

The answer is that a concept known as a *Protocol* is used.

A Protocol is an informal description of the programmer interface provided by something in Python (for example a class but it could also be a module or a set of stand-alone functions).

It is defined solely via documentation (and thus the `Calculator` class should have a class documentation string defining its protocol).

Based on the information provided by the protocol if a function or method requires an object to provide a specific operation (or method) then if it all works great; if not an error will be thrown, and the type is not compatible.

It is one of the key elements in Python which allows the Object-Oriented concept of *Polymorphism* to operate.

27.5 An Protocol Example

There are numerous commonly occurring Protocols that can be found in Python. For example, there is a protocol for defining Sequences, such as a container that can be accessed an item at a time.

This protocol requires that any type that will be held in the container must provide the `__len__()` and `__getitem__()` methods.

Thus any class that implements these two methods meets the requirements of the protocol.

However, because protocols are informal and unenforced in Python it is not actually necessary to implement all the methods in a protocol (as we saw in the previous section). For example, if it is known that a class will only be used with iteration then it may only be necessary to implement the `__getitem__()` method.

27.6 The Context Manager Protocol

Another concrete example is that of the *Context Manager Protocol*. This protocol was introduced in Python 2.5 so is very well established by now.

It is associated with the `'with as'` statement. This statement is typically used with classes which will need to allocate, and release so called *resources*.

These resources could be files, or database connections etc. In each of these cases a connection needs to be made (for example to a file or a database) before the associated object can be used.

However, the connection should then be closed and released before we finish using the object. This is because dangling connections to things such as files and databases can hang around and cause problems later on (for example typically only a limited number of concurrent connections are allowed to a file or a database at one time and if they are not closed properly a program can run out of available connections).

The `'with as'` statement ensures that any set up steps are performed before an object is available for use and that any shut down behaviour is invoked when it is finished with.

The syntax for the use of the `'with as'` statement is

```
with <managed object> as <localname>:
     Code to use managed object via <localname>
```

For example:

```
with ContextManagedClass() as cmc:
    print('In with block', cmc)
    print('Existing')
```

Note that in this case the object referenced by `cmc` is only in scope within the lines indented after the `'with as'` statement; after this the `cmc` variable is no longer accessible.

How does this work? In fact what the `'with as'` statement does is to call a special method when the `'with as'` statement is entered (just after the ':' above); this method is the `__enter__()` method. It then also calls another special method just as the `'with as'` statement is exited (just after the last indented statement). This second method is the `__exit__()` method.

- The `__enter__()` method is expected to do any setup/resource allocation/ making connections etc. It is expected to return an object that will be used within the block of statements that form that `'with as'` statement. It is common to return `self` although it is not a requirement to do so (this flexibility allows the *managed object* to act as a factory for other objects if required).
- The `__exit__()` method is called on the *managed object* and is passed information about any exceptions that might have been generated during the body of the `'with as'` statement. Note that the `__exit__()` method is called whether an exception has been thrown or not. The `__exit__()` method returns a bool, if it returns `True` then any exception that has been generated is swallowed (that is it is suppressed and not passed onto the calling code). If it returns `False` then if there is an exception it is also passed back to whatever code called the `'with as'` statement.

An example class that can be used with the `'with as'` statement (that meets the *Context Manager Protocol* requirements) is given below:

```
class ContextManagedClass(object):
    def __init__(self):
        print('__init__')

    def __enter__(self):
        print('__enter__')
        return self

    # Args exception type, exception value and traceback
    def __exit__(self, *args):
        print('__exit__:', args)
        return True

    def __str__(self):
        return 'ContextManagedClass object'
```

The above class implements the Context Manager Protocol in that it defines both the `__enter__()` method and the `__exit__()` method.

We can now use this class with the `with as` statement:

```
print('Starting')

with ContextManagedClass() as cmc:
    print('In with block', cmc)
    print('Exiting')

print('Done')
```

The output from this is:

```
Starting
__init__
__enter__
In with block ContextManagedClass object
Exiting
__exit__: (None, None, None)
Done
```

From this you can see that the `__enter__()` method is called before the code in the block and `__exit__()` is called after the code in the block.

27.7 Polymorphism

Polymorphism is the ability to send the same message (request to run a method) to different objects, each of which appear to perform the same function. However, the way in which the message is handled depends on the object's class.

Polymorphism is a strange sounding word, derived from Greek, for a relatively simple concept. It is essentially the ability to request that the same operation be performed by a wide range of different types of things. How the request is processed depends on the thing that receives the request. The programmer need not worry about how the request is handled, only that it is. This is illustrated below.

```
def night_out(p):
    p.eat()
    p.drink()
    p.sleep()
```

In this example, the parameter passed into the `night_out()` function expects to be given something that will respond to the methods `eat()`, `drink()` and `sleep()`. Any object that meets this requirement can be used with the function.

We can define multiple classes that meet this informal contract, for example we can define a class hierarchy that provides these methods, or completely separate classes that implement the methods. In the case of the class hierarchy the methods may or may not override those from the parent class.

Effectively, this means that you can ask many different things to perform the same action. For example, you might ask a range of objects to provide a printable string describing themselves. In fact in Python this is exactly what happens. For exmaple, if you ask an instance of a `Manager` class, a compiler object or a database object to return such a string, you use the same method (`__str__()`, in Python).

The name polymorphism is unfortunate and often leads to confusion. It makes the whole process sound rather grander than it actually is.

Note this is one of the most significant and flexible features of Python; it does not tie a variable to a specific type; instead via Duck Typing as long as the object provided meets the implied contract, then we are good.

The following classes all meet the contract implied by the `night_out()` function:

```
class Person:
    def eat(self): print('Person - Eat')
    def drink(self): print('Person - Drink')
    def sleep(self): print('Person - Sleep')

class Employee(Person):
    def eat(self): print('Employee - Eat')
    def drink(self): print('Employee - Drink')
    def sleep(self): print('Employee - Sleep')

class SalesPerson(Employee):
    def eat(self): print('SalesPerson - Eat')
    def drink(self): print('SalesPerson - Drink')

class Dog:
    def eat(self): print('Dog - Eat')
    def drink(self): print('Dog - Drink')
    def sleep(self): print('Dog - Sleep')
```

This means that instances of all of these classes can be used with the `night_out()` function.

Note that the `SalesPerson` class meets the implied contract partly via inheritance (the `sleep()` method is inherited from `Employee`).

27.8 The Descriptor Protocol

Another protocol is the *descriptor* protocol that was introduced back in Python 2.2.

Descriptors can be used to create what are known as *managed attributes*. A managed attribute is an object attributes that is managed (or protected) from direct access by external code via the descriptor. The descriptor can then take whatever action is appropriate such as validating the data, checking the format, logging the action, updating a related attribute etc.

The descriptor protocol defines four methods (as usual they are considered special methods and thus start with a double underbar `'__'`):

- `__get__(self, instance, owner)` This method is called when the value of an attribute is accessed. The `instance` is the instance being modified and the `owner` is the class defining the object. This method should return the (computed) attribute value or raise an `AttributeError` exception.
- `__set__(self, instance, value)` This is called when the value of an attribute is being set. The parameter `value` is the new value being set.
- `__delete__(self, instance)` Called to delete the attribute.
- `__set_name__(self, owner, name)` Called at the time the owning class `owner` is created. The descriptor has been assigned to `name`. This method was added to the protocol in Python 3.6.

The following class `Logger` implements the *Descriptor* protocol. It can therefore be used with other classes to log the creation, access and update of whatever attribute it is applied on.

```
class Logger(object):
    """ Logger class implementing the descriptor protocol"""
    def __init__(self, name):
        self.name = name

    def __get__(self, inst, owner):
        print('__get__:', inst, 'owner', owner,
              ', value', self.name, '=',
              str(inst.__dict__[self.name]))
        return inst.__dict__[self.name]

    def __set__(self, inst, value):
        print('__set__:', inst, '-', self.name, '=', value)
        inst.__dict__[self.name] = value

    def __delete__(self, instance):
        print('__delete__', instance)

    def __set_name__(self, owner, name):
        print('__set_name__', 'owner', owner, 'setting', name)
```

Each of the methods defined for the protocol prints out a message so that access can be monitored.

The `Logger` class is used with the following `Cursor` class.

```
class Cursor(object):
    # Set up the descriptors at the class level
    x = Logger('x')
    y = Logger('y')

    def __init__(self, x0, y0):
        # Initialise the attributes
        # Note use of __dict__ to avoid using self.x notation
        # which would invoke the descriptor behaviour
        self.__dict__['x'] = x0
        self.__dict__['y'] = y0

    def move_by(self, dx, dy):
        print('move_by', dx, ',', dy)
        self.x = self.x + dx
        self.y = self.y + dy

    def __str__(self):
        return 'Point[' + str(self.__dict__['x']) +
               ', ' + str(self.__dict__['y']) + ']'
```

There are several points to note about this class definition including:

The *Descriptors* must be defined at the class level not at the object/instance level. This the x and y attributes of Cursor object are defined as having Logger descriptors within the class (not within the __init__() method). If you try to define them using self.x and self.y the descriptors will not be registered.

The Cursor __init__() method uses the __dict__ dictionary to initialise the instance/object attributes x and y. This is an alternative approach to accessing an objects' attributes; it is used internally by an object to hold the actual attribute values. It by passes the normal attribute look up mechanism invoked when you use the *dot* notation (such as curser.x = 10). This means that it will not be intercepted by the Descriptor. This has been done because the logger uses the __str__() method to print out the instance holding the attribute which uses the current values of x and y. When the value of x is initially set there will be no value for y and thus an error would be generated by the __str__().

The __str__() method also uses the __dict__ dictionary to access the attributes as it is not necessary to log this access. It would also become recursive if the Logger also used the method to print out the instance.

We can now use instances of the Cursor object without knowing that the descriptor will intercept access to the attributes x and y:

```
cursor = Cursor(15, 25)
print('-' * 25)

print('p1:', cursor)
cursor.x = 20
cursor.y = 35
print('p1 updated:', cursor)
print('p1.x:', cursor.x)
print('-' * 25)

cursor.move_by(1, 1)
print('-' * 25)

del cursor.x
```

The output from this illustrates how the descriptors have intercepted access to the attributes. Note that the move_by() method accesses both the getter and the setter descriptor methods as this method reads the current value of the attributes and then updates them.

```
__set_name__ owner <class '__main__.Cursor'> setting x
__set_name__ owner <class '__main__.Cursor'> setting y
-------------------------
p1: Point[15, 25]
__set__: Point[15, 25] - x = 20
__set__: Point[20, 25] - y = 35
p1 updated: Point[20, 35]
__get__: Point[20, 35] owner <class '__main__.Cursor'> , value
x = 20
p1.x: 20
-------------------------
move_by 1 , 1
__get__: Point[20, 35] owner <class '__main__.Cursor'> , value
x = 20
__set__: Point[20, 35] - x = 21
__get__: Point[21, 35] owner <class '__main__.Cursor'> , value
y = 35
__set__: Point[21, 35] - y = 36
-------------------------
__delete__ Point[21, 36]
```

27.9 Online Resources

The following online resources focussing on Python protocols are available:

- https://ref.readthedocs.io/en/latest/understanding_python/interfaces/existing_protocols.html Documentation on Pythons default (native) protocols including the comparison, hash, attribute access and sequence protocols.
- https://docs.python.org/3/library/stdtypes.html#context-manager-types for Context manager types.
- https://pymotw.com/3/contextlib/index.html The Python Module of the Week for Context Manager Utilities.
- https://en.wikipedia.org/wiki/Polymorphism_(computer_science) Wikipedia page on Polymorphism.

27.10 Exercises

This exercise involves implementing the Context Manager Protocol.

Return to your `Account` related classes.

Modify the `Account` class such that it implements the *Context Manager* Protocol. This means that you will need to implement the `__enter__()` and `__exit__()` methods.

Place `print` messages within the methods so that you can see when they are run.

The new methods you have defined will be inherited by each of the subclasses you have created; namely `CurrentAccount`, `DepositAccount` and `InvestmentAccount`.

Now test out your modified calculator using:

```
with accounts.CurrentAccount ('891', 'Adam', 5.0, 50.0) as acc:
    acc.deposit(23.0)
    acc.withdraw(12.33)
    print(acc.balance)
```

Which should produce output similar to:

```
Creating new Account
__enter__
15.5
__exit__: (None, None, None)
```