Chapter 19
Python Classes

19.1 Introduction

In Python everything is an object and as such is an example of a type or class of things. For example, integers are an example of the int class, real numbers are examples of the float class, etc. In this chapter we will look at what a class is in Python, how it is defined, how instances of a class can be created, how attributes (data) and methods (behaviour) can be defined for a class and outline how memory is managed within Python with respect to instance creation and deletion.

19.2 Python and Classes

As mentioned above everything in Python is an object! This is illustrated below for a number of different types within Python:

```
print(type(4))
print(type(5.6))
print(type(True))
print(type('Natalia'))
print(type([1, 2, 3, 4]))
```

This prints out a list of classes that define what it is to be an `int`, or a `float` or a `bool` etc. in Python:

J. Hunt, *A Beginners Guide to Python 3 Programming*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-35122-8_19

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
<class 'list'>
```

That is the number 4 is an example or instance of the class int representing how integers work in Python. In turn the number 5.6 is an instance of the class float (representing floating-point numbers). The value True is an instance of the class bool and the string 'Natalia' is an instance of the class str (representing for strings in Python). Finally [1, 2, 3, 4] is an instance of a type of collection called a list (which is discussed later in this book).

However, you are not just restricted to the built-in types (aka classes); it is also possible to define user defined types (classes). These can be used to create your own data structures, your own data types, your own applications, etc.

Three remainder of this chapter considers the constructs in Python used to create user defined classes.

19.3 Class Definitions

In Python, a class definition has the following format

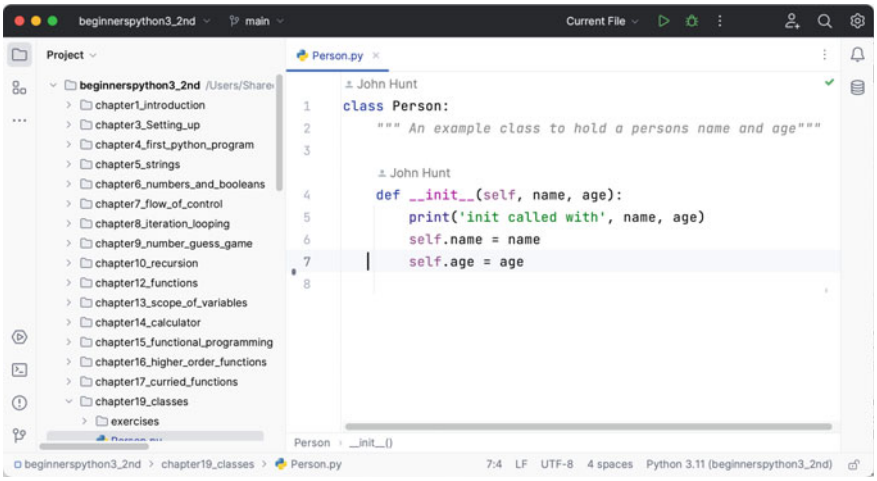
```
class nameOfClass(SuperClass):
   __init__
   attributes
   methods
```

Although you should note that you can mix the order of the definition of attributes, and methods as required within a single class.

The following code is an example of a class definition:

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

Although this is not a hard and fast rule, it is common to define a class in a file named after that class. For example, the above code would be stored in a file called `Person.py`; this makes it easier to find the code associated with a class. This is shown below using the PyCharm IDE:



The Person class possesses two *attributes* (or instance variables) called `name` and `age`.

There is also a special method defined called `__init__`. This is an initializer (also known as a constructor) for the class. It indicates what data must be supplied when an instance of the `Person` class is created and how that data is stored internally.

In this case a `name` and an `age` must be supplied when an instance of the `Person` class is created.

The values supplied will then be stored within an instance of the class (represented by the *special* variable `self`) in instance variables/attributes `self.name` and `self.age`. Note that the parameters to the `__init__` method are *local* variables and will disappear when the method terminates, but `self.name` and `self.age` are instance variables and will exist for as long as the object is available.

Let us look for a moment at the special variable *self*. This is the first parameter passed into any method. However, when a method is called we do not pass a value for this parameter ourselves; Python does. It is used to represent the object within which the method is executing. This provides the context within which the method runs and allows the method to access the data held by the object. Thus *self* is the object itself.

You may also be wondering about that term *method*. A method is the name given to behaviour that is linked directly to the `Person` class; it is not a free-standing function rather it is part of the definition of the class `Person`.

Historically, it comes from the language Smalltalk; this language was first used to simulate a production plant and a method represented some behaviour that could be used to simulate a change in the production line; it therefore represented a *method* for making a change.

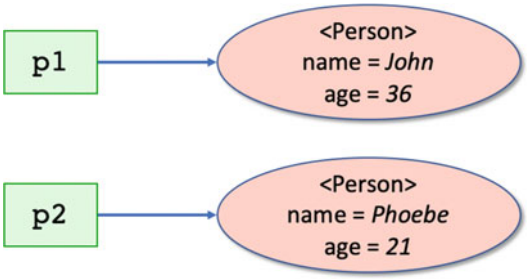
19.4 Creating Examples of the Class Person

New instances/objects (examples) of the class `Person` can be created by using the name of the class and passing in the values to be used for the parameters of the initialization method (with the exception of the first parameter *self* which is provided automatically by Python).

For example, the following creates two instances of the class Person:

```
p1 = Person('John', 36)
p2 = Person('Phoebe', 21)
```

The variable `p1` holds a reference to the *instance* or *object* of the class `Person` whose attributes hold the values 'John' (for the `name` attribute) and 36 (for the `age` attribute). In turn the variable `p2` references an instance of the class `Person` whose `name` and `age` attributes hold the values 'Phoebe' and 21. Thus in memory we have:



The two variables reference separate *instances* or examples of the class `Person`. They therefore respond to the same set of methods/operations and have the same set of attributes (such as `name` and `age`); however, they have their own values for those attributes (such as 'John' and 'Phoebe').

Each instance also has its own unique identifier—that shows that even if the attribute values happen to be the same between two objects (for example, there happen to be two people called John who are both 36); they are still separate instances of the given class. This identifier can be accessed using the `id()` function, for example:

```
print('id(p1):', id(p1))
print('id(p2):', id(p2))
```

When this code is run `p1` and `p2` will generate different identifiers, for example:

```
id(p1): 4547191808
id(p2): 4547191864
```

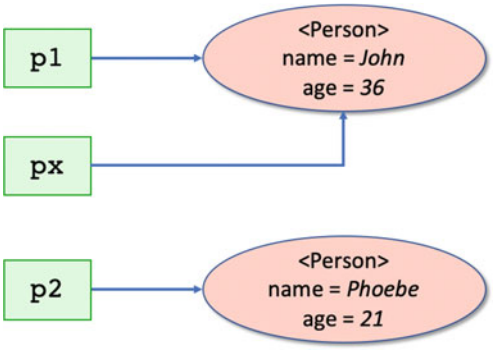
Note that actual number generated may vary from that above but should still be unique (within your program).

19.5 Be Careful with Assignment

Given that in the above example, `p1` and `p2` reference different instances of the class `Person`; what happens when `p1` or `p2` are assigned to another variable? That is, what happens in this case:

```
p1 = Person('John', 36)
px = p1
```

What does `px` reference? Actually, it makes a complete copy of the value held by `p1`; however, `p1` does not hold the instance of the class `Person`; it holds the address of the object. It thus copies the address held in `p1` into the variable `px`. This means that both `p1` and `px` now reference (point at) the same instance in memory; we there have this:



This may not be obvious when you print `p1` and `px`:

```
print(p1)
print(px)
```

As this could just imply that the object has been copied:

```
John is 36
John is 36
```

However, if we print the unique identifier for what is referenced by `p1` and `px` then it becomes clear that it is the same instance of class `Person`:

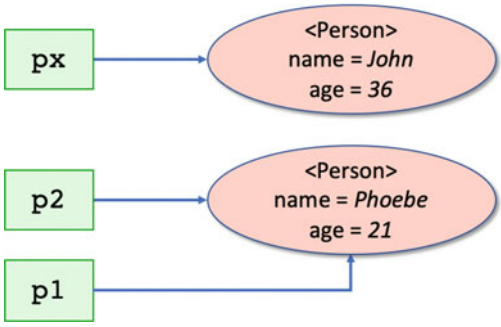
```
print('id(p1):', id(p1))
print('id(px):', id(px))
```

which prints out

```
id(p1): 4326491864
id(px): 4326491864
```

As can be seen the unique identifier is the same.

Of course, if `p1` is subsequently assigned a different object (for example, if we ran `p1 = p2`) then this would have no effect on the value held in `px`; indeed, we would now have:



19.6 Printing Out Objects

If we now use the `print()` function to print the objects held by `p1` and `p2`, we will get what might at first glance appear to be a slightly odd result:

```
print(p1)
print(p2)
```

The output generated is

```
<__main__.Person object at 0x10f08a400>
<__main__.Person object at 0x10f08a438>
```

What this is showing is the name of the class (in this case Person) and a hexadecimal number indicates where it is held in memory. Neither of which is particularly useful and certainly doesn't help us in knowing what information `p1` and `p2` are holding.

19.6.1 Accessing Object Attributes

We can access the attributes held by `p1` and `p2` using what is known as the *dot* notation. This notation allows us to follow the variable holding the object with a dot ('.') and the attribute we are interested in access. For example, to access the name of a person object we can use `p1.name` or for their age we can use `p1.age`:

```
print(p1.name, 'is', p1.age)
print(p2.name, 'is', p2.age)
```

The result of this is that we output

```
John is 36
Phoebe is 21
```

Which is rather more meaningful.

In fact, we can also update the attributes of an object directly, for example we can write:

```
p1.name = 'Bob'
p1.age = 54
```

If we now run

```
print(p1.name, 'is', p1.age)
```

then we will get

```
Bob is 54
```

We will see in a later chapter (Python Properties) that we can restrict access to these attributes by making them into *properties*.

19.6.2 Defining a Default String Representation

In the previous section we printed out information from the instances of class `Person` by accessing the attributes `name` and `age`.

However, we now needed to know the internal structure of the class `Person` to print out its details. That is, we need to know that there are attributes called `name` and `age` available on this class.

It would be much more convenient if the object itself knew how to convert its self into a string to be printed out!

In fact we can make the class Person do this by defining a method that can be used to convert an object into a string for printing purposes.

This method is the `__str__` method. The method is expected to return a string which can be used to represent appropriate information about a class.

The signature of the method is

```
def __str__(self)
```

Methods that start with a double underbar ('__') are by convention considered special in Python and we will see several of these methods later on in the book. For the moment we will focus only on the `__str__()` method.

We can add this method to our class `Person` and see how that affects the output generated when using the `print()` function.

We will return a string from the __str__ method that provides and the name and age of the person:

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

   def __str__(self):
      return self.name + ' is ' + str(self.age)
```

Note that in the __str__ method we access the name and age attributes using the self parameter passed into the method by Python. Also note that it is necessary to convert the age number attribute into a string. This is because the '+' operator will do string *concatenation* unless one of the operands (one of the sides of the '+') is a number; in which case it will try and do arithmetic addition which of course will not work if the other operand is a string!

If we now try to print out p1 and p2:

```
print(p1)
print(p2)
```

The output generated is:

```
John is 36
Phoebe is 21
```

Which is much more useful.

19.6.3 Defining a Default Storage Representation

Somewhat confusingly Python provides two ways in which an instance of a class can be converted into a String. One approach is the __str__() method described in the previous section. The other is __repr()__ method. An example of a __repr__() string representation function is given below:

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

   def __repr__(self):
      return f'Person(name = {self.name}, age = {self.age})'
```

If we now use the following code to print an instance of the class Person:

```
p1 = Person('John', 36)
print(p1)
```

The output that is generated is

```
Person(name=John, age=36)
```

So what's the difference between `__str__()` and `__repr__()`? The answer is that:

- `__str__()` is intended dot be used to create a string version of an object that can be used for printing and logging purposes.
- `__repr__()` is intended to generate a string representation of an object that can be used to recreate the object - hence the format used above that would allow you to use `Person(name = John, age = 36)` to recreate an instance of the person John.

Which is used by Python depends on what you have define. If you print an object out then Python first tries to use the `__str__()` method, if that does not exist it uses the `__repr__()` method. You can therefore choose to just use the `__repr__()` method. However, if instances of a class are contained within for example a list, then when the list is printed out it will always use the `__repr__()` method. For this reason it is quite common to find that a class has both a `__str__()` and a `__repr__()` method using slightly different formats. Alternative, one of the two methods may call the other one.

For example we might define the class `Person` as:

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

   def __init__(self, name, age):
      print('init called with', name, age)
      self.name = name
       self.age = age

   def __str__(self):
      return self.name + ' is ' + str(self.age).

   def __repr__(self):
       return f'Person(name = {self.name}, age = {self.age})'
```

19.7 Providing a Class Comment

It is common to provide a comment for a class defining what that class does, its purpose and any important points to note about the class.

This can be done by providing a *docstring* for the class just after the class declaration header; you can use the triple quotes string (`"""..."""`) to create multiple line *docstrings*, for example:

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

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

   def __str__(self):
      return self.name + ' is ' + str(self.age)
```

The *docstring* is accessible through the `__doc__` attribute of the class. The intention is to make information available to users of the class, even at runtime. It can also be used by IDEs to provide information on a class.

Note that a class comment can also contain reStructured Text formatting commands and directives. This is a common practice as it provides improved information layout over a basic comment.

19.8 Adding a Birthday Method

Let us now add some behaviour to the class `Person`. In the following example, we define a *method* called `birthday()` that takes no parameters and increments the `age` attribute by 1:

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

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

   def __str__(self):
      return self.name + ' is ' + str(self.age)

   def birthday(self):
      print ('Happy birthday you were', self.age)
      self.age += 1
      print('You are now', self.age)
```

Note that again the first parameter passed into the method `birthday` is `self`. This represents the instance (the example of the class `Person`) that this method will be used with.

If we now create an instance of the class `Person` and call `birthday()` on it, the `age` will be incremented by 1, for example:

```
p3 = Person('Adam', 19)
print(p3)
p3.birthday()
print(p3)
```

When we run this code, we get

```
Adam is 19
Happy birthday you were 19
You are now 20
Adam is 20
```

As you can see Adam is initially 19; but after his birthday he is now 20.

19.9 Defining Instance Methods

The `birthday()` method presented above is an example of what is known as an instance method; that is, it is tied to an instance of the class. In that case the method did not take any parameters, nor did it return any parameters; however, instance methods can do both.

For example, let us assume that the `Person` class will also be used to calculate how much someone should be paid. Let us also assume that the rate is £7.50 if you are under 21 but that there is a supplement of 2.50 if you are 21 or over.

We could define an instance method that will take as input the number of hours worked and return the amount someone should be paid:

```
class Person:
   """ An example class to hold a persons name and age"""
   #...
    def calculate_pay(self, hours_worked):
       rate_of_pay = 7.50
       if self.age >= 21:
          rate_of_pay += 2.50
       return hours_worked * rate_of_pay
```

We can invoke this method again using the *dot* notation, for example:

```
pay = p2.calculate_pay(40)
print('Pay', p2.name, pay)
pay = p3.calculate_pay(40)
print('Pay', p3.name, pay)
```

Running this shows that Phoebe (who is 21) will be paid £400 while Adam who is only 19 will be paid only £300:

```
Pay Phoebe 400.0
Pay Adam 300.0
```

Another example of an instance method defined on the class Person is the `is_teenager()` method. This method does not take a parameter, but it does return a Boolean value depending upon the age attribute:

```
class Person:
    """ An example class to hold a persons name and age"""
   #...
    def is_teenager(self):
       return self.age < 20
```

Note that the *implicitly* provided parameter `'self'` is still provided even when a method does not take a parameter.

19.10 Person Class Recap

Let us bring together the concepts that we have looked at so far in the final version of the class `Person`.

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

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

    def __str__(self):
           return self.name + ' is ' + str(self.age)

    def birthday(self):
       print ('Happy birthday you were', self.age)
       self.age += 1
       print('You are now', self.age)

    def calculate_pay(self, hours_worked):
       rate_of_pay = 7.50
       if self.age >= 21:
          rate_of_pay += 2.50
       return hours_worked * rate_of_pay

    def is_teenager(self):
       return self.age < 20
```

This class exhibits several features we have seen already and expands a few others:

- The class has a two-parameter initializer that takes a `String` and an `Integer`.
- It defines two attributes held by each of the instances of the class; `name` and `age`.
- It defines a `__str__` method so that the details of the Person object can be easily printed.

- It defines three methods `birthday()`, `calculate_pay()` and `is_teenager()`.
- The method `birthday()` does not return anything (i.e., it does not return a value) and is comprised of three statements, two print statements and an assignment.
- `is_teenager()` returns a `Boolean` value (i.e., one that returns `True` or `False`).

An example application using this class is given below:

```
p1 = Person('John', 36)
print(p1)
print (p1.name, 'is', p1.age)
print('p1.is_teenager', p1.is_teenager())
p1.birthday()
print(p1)
p1.age = 18
print(p1)
```

This application creates an instance of the `Person` class using the values 'John' and 36. It then prints out `p1` using `print` (which will automatically call the `__str__()` method on the instances passed to it). It then accesses the values of name and age properties and prints these. Following this it calls the `is_teenager()` method and prints the result returned. It then calls the `birthday()` method. Finally, it assigns a new value to the `age` property. The output from this application is given below:

```
John is 36
John is 36
p1.is_teenager False
Happy birthday you were 36
You are now 37
John is 37
John is 18
```

19.11 The Del Keyword

Having at one point created an object of some type (whether that is a `bool`, an `int` or a user defined type such as `Person`) it may later be necessary to delete that object. This can be done using the keyword `del`. This keyword is used to delete objects which allows the memory they are using to be reclaimed and used by other parts of your program.

For example, we can write

```
p1 = Person('John', 36)
print(p1)
del p1.
```

After the `del` statement the object held by `p1` will no longer be available and any attempt to reference it will generate an error.

You do not need to use `del` as setting `p1` above to the `None` value (representing nothingness) will have the same effect. In addition, if the above code was defined within a function or a method then `p1` will cease to exist once the function or method terminates and this will again have the same effect as deleting the object and freeing up the memory.

19.12 Automatic Memory Management

The creation and deletion of objects (and their associated memory) is managed by the Python Memory Manager. Indeed, the provision of a memory manager (also known as automatic memory management) is one of Python's advantages when compared to languages such as C and C++. It is not uncommon to hear C++ programmers complaining about spending many hours attempting to track down a particularly awkward bug only to find it was a problem associated with memory allocation or pointer manipulation. Similarly, a regular problem for C++ developers is that of memory creep, which occurs when memory is allocated but is not freed up. The application either uses all available memory or runs out of space and produces a run time error.

Most of the problems associated with memory allocation in languages such as C++ occur because programmers must not only concentrate on the (often complex) application logic but also on memory management. They must ensure that they allocate only the memory which is required and deallocate it when it is no longer required. This may sound simple, but it is no mean feat in a large complex application.

An interesting question to ask is "why do programmers have to manage memory allocation?". There are few programmers today who would expect to have to manage the registers being used by their programs, although 30 or 40 years ago the situation was very different. One answer to the memory management question, often cited by those who like to manage their own memory, is that "it is more efficient, you have more control, it is faster and leads to more compact code". Of course, if you wish to take these comments to their extreme, then we should all be programming in assembler. This would enable us all to produce faster, more efficient and more compact code than that produced by Python or languages such as Java.

The point about high level languages, however, is that they are more productive, introduce fewer errors, are more expressive and are efficient enough (given modern computers and compiler technology). The memory management issue is somewhat similar. If the system automatically handles the allocation and deallocation of memory, then the programmer can concentrate on the application logic.

This makes the programmer more productive, removes problems due to poor memory management and, when implemented efficiently, can still provide acceptable performance.

Python therefore provides automatic memory management. Essentially, it allocates a portion of memory as and when required. When memory is short, it looks for areas which are no longer referenced. These areas of memory are then freed up (deallocated) so that they can be reallocated. This process is often referred to as *Garbage Collection.*

19.13 Intrinsic Attributes

Every class (and every object) in Python has a set of *intrinsic* attributes set up by the Python runtime system. Some of these intrinsic attributes are given below for classes and objects.

Classes have the following intrinsic attributes:

- `__name__` the name of the class
- `__module__` the module (or library) from which it was loaded
- `__bases__` a collection of its base classes (see inheritance later in this book)
- `__dict__` a dictionary (a set of key-value pairs) containing all the attributes (including methods)
- `__doc__` the documentation string

For objects:

- `__class__` the name of the class of the object
- `__dict__` a dictionary containing all the object's attributes

Notice that these intrinsic attributes all start and end with a double underbar—this indicates their special status within Python.

An example of printing these attributes out for the class `Person` and a instance of the class are shown below:

```
print('Class attributes')
print(Person.__name__)
print(Person.__module__)
print(Person.__doc__)
print(Person.__dict__)
print('Object attributes')
print(p1.__class__)
print(p1.__dict__)
```

The output from this is:

```
Class attributes
Person
__main__
An example class to hold a persons name and age
```

```
{'__module__': '__main__', '__doc__': ' An example class to hold a
persons name and age', 'instance_count': 4, 'increment_instance_
count': <classmethod object at 0x105955588>, 'static_function':
<staticmethod object at 0x1059555c0>, '__init__': <function
Person.__init__ at 0x10595d268>, '__str__': <function Person._
_str__ at 0x10595d2f0>, 'birthday': <function Person.birthday
at 0x10595d378>, 'calculate_pay': <function Person.calculate_
pay at 0x10595d400>, 'is_teenager': <function Person.is_teenager
at 0x10595d488>, '__dict__': <attribute '__dict__' of 'Person'
objects>, '__weakref__': <attribute '__weakref__' of 'Person'
objects>}
Object attributes
<class '__main__.Person'>
{'name': 'John', 'age': 36}
```

19.14 Online Resources

See the following for further information on Python classes:

- https://docs.python.org/3/tutorial/classes.html the Python Standard library Class tutorial.
- https://www.tutorialspoint.com/python3/python_classes_objects.htm The tutorials point tutorial on Python 3 classes.

19.15 Exercises

The aim of this exercise is to create a new class called `Account`.

1. Define a new class to represent a type of bank account.
2. When the class is instantiated you should provide the account number, the name of the account holder, an opening balance and the type of account (which can be a string representing 'current', 'deposit' or 'investment', etc.). This means that there must be an __init__ method and you will need to store the data within the object.
3. Provide three instance methods for the `Account`; `deposit(amount)`, `withdraw(amount)` and `get_balance()`. The behaviour of these methods should be as expected, deposit will increase the balance, withdraw will decrease the balance and get_balance() returns the current balance.
4. Define a simple test application to verify the behaviour of your Account class.

It can be helpful to see how your class Account is expected to be used. For this reason a simple test application for the Account is given below:

```
acc1 = Account('123', 'John', 10.05, 'current')
acc2 = Account('345', 'John', 23.55, 'savings')
```

```
acc3 = Account('567', 'Phoebe', 12.45, 'investment')

print(acc1)
print(acc2)
print(acc3)

acc1.deposit(23.45)
acc1.withdraw(12.33)
print('balance:', acc1.get_balance())
```

The following output illustrates what the result of running this test application might look like:

```
Account[123] - John, current account = 10.05
Account[345] - John, savings account = 23.55
Account[567] - Phoebe, investment account = 12.45
balance: 21.17
```