OOP IN PYTHON

12.1 Overview of OOP Principles

Object Oriented Programming (*OOP*) is a programming paradigm – a way of looking at problems and designing solutions. The main goal behind OOP is to model the real world within the software so that we have a parallel reality, making it easier for us to relate code to the real world. This programming paradigm essentially uses the following 8 principles to achieve this goal:

1. Classes
2. Objects
3. Data Encapsulation
4. Data Hiding
5. Data Abstraction
6. Polymorphism
7. Inheritance
8. Message Passing

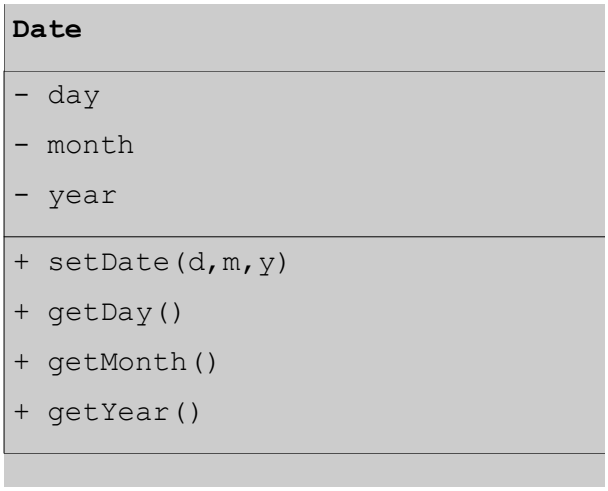
We therefore will briefly cover these 8 principles before proceeding with how OOP can be used in Python.

12.1.1 Class

A *class* can be defined as a design according to which objects can be later instantiated.

The starting point of Object Oriented Modelling is the class. The class is the design of an entity that exists in the real world. This design comprises of attributes (everything an entity has) and behaviour (everything an entity can do).

As an example, here is a diagrammatic representation of a `Date` class (technically using UML for the class representation) to represent a calendar date:



Note:

1. For now, it is sufficient to just note that `Date` is the name of a class, which has 3 attributes – `day`, `month` and `year` – and 4 member functions – `setDate()`, `getDay()`, `getMonth()` and `getYear()`.
2. Thus, the 3 pieces within the above UML diagram are: class name, data members and member functions.
3. The "`-`" prefix denotes that we want to keep these members private (section 12.4.6.2)
4. The "`+`" prefix similarly denotes that we want to keep these members public (again, section 12.4.6).

12.1.2 Object

An *object* is an instance of a class.

Once a class has been completely designed, multiple individual instances of the class can be created just as how once a car has been designed, multiple cars can be manufactured of the same design. These objects are identical to each other in terms of their design and yet independent of each other in terms of the values of their attributes. Thus, the different cars that are manufactured out of the same design can have different values for their attributes like their colour. The behaviour of all objects of the same class have to be identical, however. Thus, objects of a class typically have the same attributes and behaviour, but can differ from each other in the values of their attributes.

In our `Date` example, `day`, `month` and `year` are attributes whereas `setDate()`, `getDay()`, `getMonth()` and `getYear()` are behaviour. Multiple `Date` objects can have different values for `day`, `month` and `year`, thereby representing possibly different dates, but will have identical functionality.

12.1.3 Data Encapsulation

Data Encapsulation refers to the encapsulation of data and the code that acts on the data into a single unit. Since a class packs together attributes (data) and behaviour (code that acts on the data), we claim that classes make data encapsulation possible.

Data Encapsulation is important in the programming world as it provides an equal status for code and data.

In our `Date` example, the `Date` class encapsulates both the data (`day`, `month` and `year`) and the code that acts on the data (`setDate()`, `getDay()`, `getMonth()` and `getYear()`).

12.1.4 Data Hiding

An unwritten law in OOP is that all objects have to relate to reality at all times. Since an object in an instance of a class and a class is the design of an entity in the real world, we expect every object in our program to represent a valid real-world entity at all times. This law also entails ensuring that the attributes of an object are never meddled with in a way that it represents something absurd in reality. One of the ways of enforcing this is by hiding data that belongs to an object from the external world so that no one can accidentally tamper with it – a concept called *Data Hiding*.

In our `Date` example, we could try to hide the attributes `day`, `month` and `year` so as to prevent the users from tampering with them directly. This would prevent the user from, say, setting an invalid date like `40/30/2016`! We will instead provide functions for setting and getting the date, with validations enforced as necessary.

12.1.5 Data Abstraction

Data Abstraction refers to the hiding of implementation details while revealing a simple interface. The first advantage of Data Abstraction is that it makes the object easy to use as the users need to only be aware of the interface and need not know any implementation details. Secondly, when the implementation is insulated from the usage via the interface, it makes it possible to later change the implementation without affecting the usage by keeping the interface consistent. This adds a lot of value in practical programming, especially during the maintenance of the software.

In our `Date` example, we have provided a simple interface to set and get the date without the user having to know how we end up storing the date within the object.

Data Hiding and Data Abstraction are very much related but are not exactly the same! Both concentrate on hiding something, but with very different intentions! While Data Hiding concentrates on hiding data in order to help maintain the integrity of the object, Data Abstraction concentrates on hiding implementation details in order to simplify the usage of the object.

12.1.6 Polymorphism

The term *polymorphism* in general means multiple forms of the same entity. In OOP, it means performing the same logical operation by choosing from multiple implementations appropriately. The two forms of polymorphism that we are going to see actively in Python are:

1. *Operator overloading* – performing the same logical operation using an operator in different ways (using different implementation) depending on the type of the invoking operand
2. *Dynamic polymorphism* – performing the same logical operation in different ways (using different implementation) depending on the type of the invoking object.

In the `Date` example, an example of operator overloading would be providing an implementation for addition of days to a given date, resulting in a new Date object.

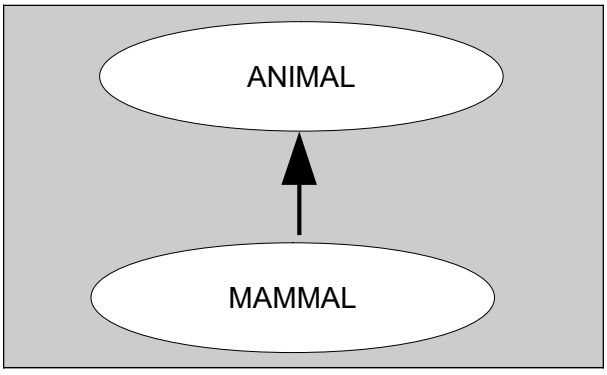
12.1.7 Inheritance

Inheritance is the mechanism wherein a class acquires all the features and properties of another class (or classes). Inheritance has the following uses:

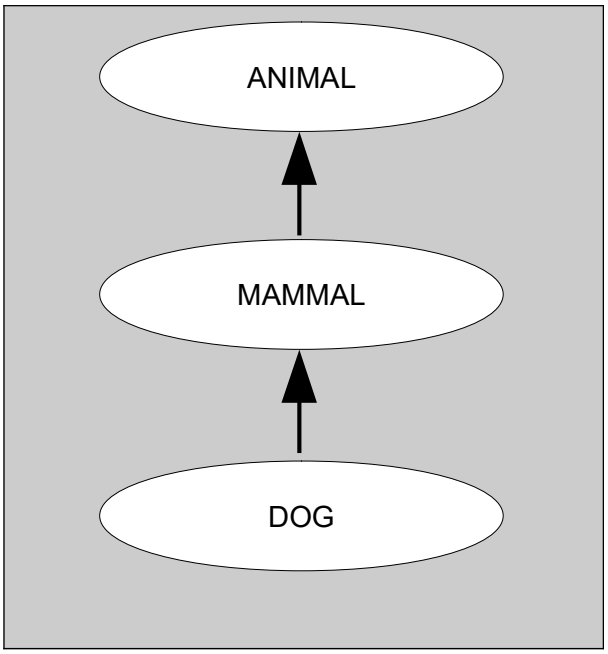
1. **Re-usability** – it helps reduce effort by reusing existing code.
2. **Extensibility** – it helps add new code or apply changes without tampering with existing code.
3. **Compartmentalisation** – it helps manage code better by compartmentalising classes.

Structurally, there are multiple types of inheritance:

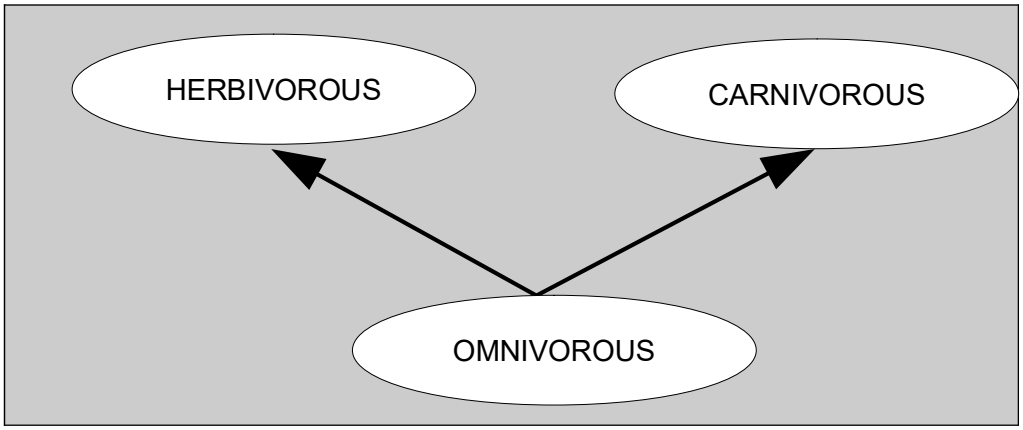
1. **Simple inheritance** – when a class inherits from another class.



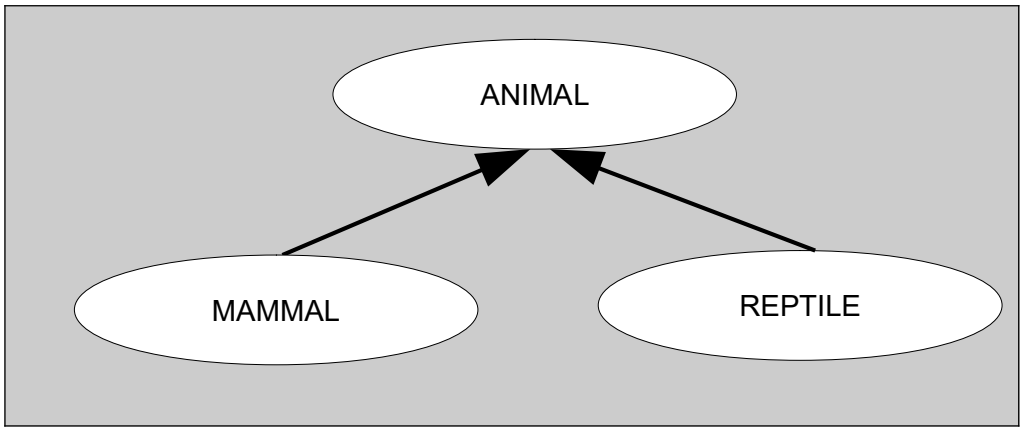
2. **Multi-level inheritance** – when a class inherits from a class that inherits from another class, thereby forming an inheritance chain.



3. **Multiple inheritance** – when a class inherits from multiple classes.



4. **Hierarchical inheritance** – When multiple classes inherit from a single class.



Inheritance is an important concept as dynamic polymorphism is totally dependent on it. In our `Date` example, we can inherit custom date format classes like `USDate` and `UKDate` from the `Date` class, adding only as much functionality as needed to support the date format.

12.1.8 Message Passing

In OOP, programs are expected to document class structures, create objects at runtime and permit these objects to communicate with each other at runtime. This inter-object communication between objects at runtime is called *message passing* and is implemented typically by making function calls, with the arguments passed being technically called the *message*.

12.2 Defining Classes

Designing a class is the first step in OOP. The simplest syntax of defining a class in Python is shown below:

```
class className:
    statements
```

The statements within a class can be any of the following:

1. Blank lines
2. Comments
3. Class variables
4. Class functions

We will cover class variables in section 12.4.3 and class functions in section 12.4.4. For now, let us start with an empty class. Since at least 1 statement is compulsory within a class definition (similar to how conditions, loops and functions need to have at least 1 statement within their body), we can use the `pass` statement to get away without causing any effect:

```
class Date:
      pass
```

Just as how a function definition can optionally start with a documentation string, even class definitions can contain documentation strings, which are then accessible using the special class variable `__doc__`:

```
>>> class Date:
...     """This is our first implementation of the Date class"""
...     pass
...
>>> Date.__doc__
'This is our first implementation of the Date class'
```

We will add more code into the class definition as we learn more features in Python.

12.3 Instantiating Classes

Once our class definition is ready, we can instantiate it and create objects. Each object will have everything documented in the class definition. In our `Date` example, since the class is empty, even the objects will be almost empty. Do note that the documentation string which was part of the class is now also accessible via the object.

The simplest syntax for instantiating a class and creating an object and storing it's reference in a variable is shown below:

```
var=ClassName()
```

This is how we can instantiate our `Date` class:

```
>>> d=Date()
>>> d
<__main__.Date object at 0x7f592e29fb50>
>>> type(d)
<class '__main__.Date'>
>>> d.__doc__
'This is our first implementation of the Date class'
```

As can be seen, the `__doc__` member of the class is accessible through the object `d` too.

12.4 Instance Variables, Class Variables, Functions and Methods

The attributes present in a class are called *Class Variables* and the functionality provided by the class using functions are called *Class Functions*.

Class variables belong to the class and are accessible using any object of the class too. In other words, class variables could be said to be shared by objects of that class.

Objects can have their own independent variables called *Instance Variables*.

The functions defined in a class are *Class Functions* and as such belong to the class and are also accessible and invokable by any object of that class, with the object playing no other role apart from identifying the class to which the Class Function belongs. When these functions work on a per-object basis (by keeping track of the invoking object and implicitly working on it), they are called *Methods*.

This section focuses on these 4 concepts:

1. Instance Variables
2. Instance Methods
3. Class Variables
4. Class Functions

12.4.1 Instance Variables

Instance Variables are variables that belong to an object.

> **For C/C++/Java programmers:**
>
> Unlike languages like C++ and Java where the class definition specifies the instance variables, in Python the class can only specify the Class Variables (which are like `static` data members in C++ and `static` fields in Java).

Python uses the Perl style, wherein the object can create whatever instance variables it desires. In fact, it is possible for different objects of the same class to have different instance variables, though doing so might not be a good idea! This creation of instance variables is typically done using methods, as will be demonstrated in an example soon.

Instance variables can also be deleted using the `del` statement!

In our `Date` example, we want the `Date` objects to have `day`, `month` and `year` as instance variables – we want each `Date` object to have it's own value for these. These instance variables can be created in the `setDate()` method as shown below:

```
>>> class Date:
...     def setDate(self,d,m,y):
...             self.day,self.month,self.year = d,m,y
...
>>> d=Date()
>>> d.setDate(1,2,2000)
>>> d
<__main__.Date object at 0x7f592df6cb50>
>>> d.day
1
>>> d.month
2
>>> d.year
2000
```

Observation:

1. `setDate()` is a method – a function of the class `Date` that requires an invoking object to invoke it upon itself. In our example, the statement `d.setDate(1,2,2000)` shows how the `Date` object `d` invokes the function `setDate()` upon itself, passing `1`, `2` and `2000` as arguments to the method.
2. A reference to the invoking object is passed implicitly as the first argument to the method `setDate()`, and is received as the parameter `self`. While the parameter name should not matter strictly speaking, as a convention it is always named `self`. It would be wise to follow this convention.
3. Everything that belongs to the invoking object should be explicitly preceded by the reference `self`. Thus, to access the instance variable `day` of the invoking object, we would need to access `self.day`. The first time an assignment is made to an instance variable that does not exist, the instance variable is created.
4. The `setDate()` method copies the given parameters (`d`, `m` and `y`) to the corresponding instance variables (`self.day`, `self.month` and `self.year` respectively). We will improve this method later by adding validation support.
5. The above example shows that the `setDate()` works as expected.

12.4.2 Instance Methods

In the previous example, we have already seen the method `setDate()` in action. We have seen that methods are functions of the class that use invoking objects, and can access their corresponding invoking objects using the first parameter – typically called `self`.

Let us add more instance methods to our `Date` class:

```
>>> class Date:
...     def setDate(self,d,m,y):
...             self.day,self.month,self.year = d,m,y
...     def getDay(self): return self.day
...     def getMonth(self): return self.month
...     def getYear(self): return self.year
...
>>> d = Date()
>>> d.setDate(1,2,2000)
>>> print("{}-{}-{}".format(d.getDay(),d.getMonth(),d.getYear()))
1-2-2000
```

Observation:

1. We had earlier defined the method `setDate()`. Such methods that are used to set the state of an object are called *setters* or *setter methods*.
2. We have now added the methods `getDay()`, `getMonth()` and `getYear()` to obtain or extract the state of an object. Such methods are called *getters* or *getter methods*. Since these methods have a single simple statement within their bodies, they have been defined in a single line for brevity.
3. Observe that all methods (inclusive of and not limited to getters and setters) use the first parameter (`self`) to reference the invoking object.
4. After the class definition is completed, we create an object `d`, set it's date using our setter `setDate()` and extract the values back using our getters (`getDay`, `getMonth` and `getYear`) and print them to verify the whole process.

It would be a good idea to delegate the printing of the `Date` object to the object itself via a method `print()`:

```
>>> class Date:
...     def setDate(self,d,m,y):
...             self.day,self.month,self.year = d,m,y
...     def getDay(self): return self.day
...     def getMonth(self): return self.month
...     def getYear(self): return self.year
...     def print(self):
...             print("{}-{}-{}".format
(self.getDay(),self.getMonth(),self.getYear()))
...
>>> d = Date()
>>> d.setDate(1,2,2000)
>>> d.print()
1-2-2000
```

Observation:

1. We have added a method `print()` to the `Date` class to print out the date. This `print()` method has no connection to the global built-in `print()` function. We can of course think of a different name than `print()` if required.
2. By delegating the responsibility of printing to the object, the usage of the object becomes simpler!
3. We will see a better technique of achieving this objective of printing a string representation of an object in section 12.8.2.2.

The reason Data Hiding is considered important is that it eliminates accidental changes to data, thereby maintaining the integrity of the object. This is made possible by preventing direct access to data and providing accessible methods that permit assignment to non-accessible data after due validation. How data can be made inaccessible will be dealt with in section 12.4.6, but right now we will focus on providing a method that performs validation before assigning values to data. Here is the new code with the setter method `setDate()` modified to include validation:

```
>>> class Date:
...     def setDate(self,d,m,y):
...             if self.isValid(d,m,y):
...                     self.day,self.month,self.year = d,m,y
...             else:
...                     print("Invalid date!")
...     def getDay(self): return self.day
...     def getMonth(self): return self.month
...     def getYear(self): return self.year
...     def print(self):
...             print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
...     def isValid(self,d,m,y):
...             if y<1 or y>9999: return False
...             if m<1 or m>12: return False
...             daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
...             if self.isLeap(y): daysInMonth[2]=29
...             if d<1 or d>daysInMonth[m]: return False
...             return True
...     def isLeap(self,y): return y%4==0 and (not y%100==0 or y
%400==0)
...
>>> d=Date()
>>> d.setDate(1,2,2000)
>>> d.setDate(29,2,2001)
Invalid date!
```

Observation:

1. The `setDate()` method performs validation of the given inputs using the `isValid()` method (defined later in the class). Only if the given values represents a valid date, it is stored in the instance variables; otherwise an error message is displayed. Instead of displaying an error message, a more professional approach would be to throw an exception. This better version is covered in section 13.
2. The `isValid()` method first verifies that the year is between 1 and 9999 and the month is between 1 and 12. It then builds the array `daysInMonth` to keep track of the maximum days in each month. We also take into consideration the fact that leap years have 29 days in February. The day is then validated. The method simply returns `True` if the given date is valid and `False` otherwise.
3. The `isLeap()` method checks if the given year is leap or not and returns

`True` if leap and `False` otherwise. The condition for a year to be leap is that it must be divisible by 4, and if divisible by 100 then must also be divisible by 400.

4. We will see a better way of implementing the `isValid()` and `isLeap()` methods in section 12.4.4.

Before proceeding to the next section, let's add some more functionality to our `Date` class and store it as a program. We will add a mechanism to add days to a `Date` object and obtain the date after the addition:

`Date1.py`

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       def setDate(self,d,m,y):
7.           if self.isValid(d,m,y):
8.               self.day,self.month,self.year = d,m,y
9.           else:
10.              print("Invalid date!")
11.
12.      def getDay(self): return self.day
13.      def getMonth(self): return self.month
14.      def getYear(self): return self.year
15.
16.      def print(self):
17.          print("{}-{}-{}".format
(self.getDay(),self.getMonth(),self.getYear()))
18.
19.      def isValid(self,d,m,y):
20.
daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
21.          if y<1 or y>9999: return False
22.          if self.isLeap(y): daysInMonth[2]=29
23.          if m<1 or m>12: return False
24.          if d<1 or d>daysInMonth[m]: return False
25.          return True
26.
27.      def isLeap(self,y): return y%4==0 and (not y%100==0 or
y%400==0)
28.
29.      def addDays(self,days):
30.          d,m,y = self.day,self.month,self.year
31.
daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
```

```
32.          if self.isLeap(y): daysInMonth[2]=29
33.
34.          for i in range(days):
35.              d=d+1
36.              if d>daysInMonth[m]:
37.                  d=1
38.                  m=m+1
39.                  if m>12:
40.                      m=1
41.                      y=y+1
42.                      if self.isLeap(y): daysInMonth[2]=29
43.                      else: daysInMonth[2]=28
44.          result = Date()
45.          result.setDate(d,m,y)
46.          return result
47.
48.  d1 = Date()
49.  d1.setDate(1,2,2000)
50.  d2 = d1.addDays(100)
51.  d2.print()
```

Output:

```
11-5-2000
```

Observation:

1. We have introduced an `addDays()` method in line 29.
2. The `addDays()` method uses a loop (line 34) to increment the day (`d`) by 1 (line 35) as many times as the number of days to be added.
3. Whenever the day (`d`) crosses the number of days in that month (tested in line 36), we reset the day to 1 and increment the month.
4. Whenever the month crosses 12 (tested in line 39), we reset the month to 1 and increment the year.
5. Whenever the year changes, we recompute the number of days in February depending on whether the year is leap or not (lines 42-43).
6. For simplicity, we have not checked if the year crosses 9999.
7. Finally, the method returns a `Date` object from the values in `d`, `m` and `y`.
8. From the output, we can confirm that the date 100 days after 1-2-2000 is indeed 11-5-2000.

12.4.3 Class Variables

As already introduced, class variables are variables that belong to the class and are shared across instances of that class. They are accessible using the class name (the class object) or any instance of that class. The following code snippets demonstrate these features:

```
>>> class A:
...     x=10
...
>>> A.x
10
```

We have created a class `A` with a class variable `x` initialized to `10`. We can access this using `A.x`, where `A` is the class object (class name for us) and `x` is the class variable. Let us create objects now:

```
(continuation)
>>> a=A()
>>> b=A()
>>> a.x
10
>>> b.x
10
```

We have created 2 objects `a` and `b`, and can see that they both have access to the class variable `x`, and provide us the same value `10`. Let us attempt to make an assignment to the class variable using an instance:

```
(continuation)
>>> a.x=20
>>> A.x
10
>>> a.x
20
>>> b.x
10
```

When an assignment is made to a variable using an instance, the variable is assumed to be an instance variable. Thus, the statement `a.x=20` ends up creating an instance variable `x` in the instance `a` and assigns the value `20` to it without disturbing the class variable `x` that is accessible via `A` as well as `b`. Any attempt to access the variable `x` using the instance a will give preference to the instance variable `x` in the instance `a`. Let us now make an assignment to the class variable using the class object:

```
(continuation)
>>> A.x=30
>>> A.x
30
>>> a.x
20
>>> b.x
30
```

We can see that when we change the class variable using the class object, the change is visible using the class object as well as all instances of that class, except those instances that also have an instance variable with the same name (in our case, the instance variable `x` in the instance `a`).

Let us apply this concept to our `Date` class in `Date1.py`. We see that the methods `isValid` and `addDays` require the same array `daysInMonth`. They are specified twice – in lines 20 and 31. Let us make this variable a class variable, accessible by any object – and therefore any method within the object. We could of course have made this an instance variable, but the fact that the values of this array does not change from instance to instance justifies making this a class variable instead.

Date2.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def setDate(self,d,m,y):
9.           if self.isValid(d,m,y):
10.              self.day,self.month,self.year = d,m,y
11.          else:
12.              print("Invalid date!")
13.
14.      def getDay(self): return self.day
15.      def getMonth(self): return self.month
16.      def getYear(self): return self.year
17.
18.      def print(self):
19.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
20.
21.      def isValid(self,d,m,y):
22.          Date.daysInMonth[2]=28
23.          if y<1 or y>9999: return False
```

```
24.          if self.isLeap(y): Date.daysInMonth[2]=29
25.          if m<1 or m>12: return False
26.          if d<1 or d>Date.daysInMonth[m]: return False
27.          return True
28.
29.      def isLeap(self,y): return y%4==0 and (not y%100==0 or
y%400==0)
30.
31.      def addDays(self,days):
32.          d,m,y = self.day,self.month,self.year
33.          Date.daysInMonth[2]=28
34.          if self.isLeap(y): Date.daysInMonth[2]=29
35.
36.          for i in range(days):
37.              d=d+1
38.              if d>Date.daysInMonth[m]:
39.                  d=1
40.                  m=m+1
41.                  if m>12:
42.                      m=1
43.                      y=y+1
44.                      if self.isLeap(y):
Date.daysInMonth[2]=29
45.                      else: Date.daysInMonth[2]=28
46.          result = Date()
47.          result.setDate(d,m,y)
48.          return result
49.
50.  d1 = Date()
51.  d1.setDate(1,2,2000)
52.  d2 = d1.addDays(100)
53.  d2.print()
```

Observation:

1. This code is basically taken from `Date.py`. The local variable `daysInMonth` has been removed from the methods `isValid` and `addDays` and instead made a class variable in line 6.
2. Each reference to `daysInMonth` elsewhere in the code has been replaced with `Date.daysInMonth` since it is a class variable now as opposed to a local variable earlier.
3. Since the class variable `daysInMonth` is shared by all instances and the methods `isValid` and `addDays`, any change made by one of them will be visible in all others. We therefore want to be careful about the number of days in February. The addition of lines 22 and 33 makes this code reliable and resilient to past changes to the number of days in February by any method.

12.4.4 Class Functions

Just as how variables defined in a class automatically become class variables, functions defined within a class also become class functions. They are methods only when an object is used to invoke the function, in which case a reference to the invoking object is passed automatically as the first argument and is typically received as the `self` parameter. Class functions can be invoked only through the class object and obviously do not receive any invoking object reference as `self`.

The following code snippets will help demonstrate these features:

```
>>> class A:
...     x=10
...     def increment():
...             A.x=A.x+1
...
>>> A.x
10
>>> A.increment()
>>> A.x
11
```

We define a class `A` with a class variable `x` that is initialized to `10`. We define a function `increment()` that increments the value of this class variable. The function is invoked using the class object (`A`) and not by using any object of the class `A`.

```
(continuation)
>>> a=A()
>>> b=A()
>>> a.x
11
>>> b.x
11
```

We then create 2 objects of the class A (`a` and `b`) and observe that they too give us the incremented value of the class variable `x`.

We cannot however invoke the class function `increment()` using any of the objects `a` and `b` as class functions are not methods:

```
(continuation)
>>> a.increment()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: increment() takes 0 positional arguments but 1 was given
```

Class functions are preferred over methods when it is obvious that the function does not depend on any particular instance of that class. In our `Date` example, we observe that the functions `isValid()` and `isLeap()` are in no way connected to any specific instance of the `Date` class, and are thus candidates for conversion into class functions.

For beginners, a simple technique to determine when to convert instance methods to class functions is when there is no usage of `self` within the function definition, except for accessing other methods which are also candidates for conversion to class functions. In our `Date` example, the method `isValid()` does use `self` to invoke the method `isLeap()`. The `isLeap()` method does not use `self` in it's definition and hence can be converted to a class function. After this change, the call to `isLeap()` within `isValid()` changes from `self.isLeap()` to `Date.isLeap()` and there is no usage of `self` within `isValid()`, making it possible to convert `isValid()` also from an instance method to a class function.

In case you are wondering why we should convert these instance methods into class functions when the code is working perfectly fine, it is because it is artificial and could also be misleading or meaningless. In the statement `self.isValid(d,m,y)`, `self` has absolutely no role to play! Similarly, in the statement `d1.isLeap(2000)`, `d1` has no role to play and could be misleading at worst and meaningless at best. It is far better to replace them with `Date.isValid(d,m,y)` and `Date.isLeap(2000)` respectively. Also note that class functions can be invoked even when the class has not been instantiated while instance methods cannot be invoked without an instance.

NOTE:

Python 2.2 and above supports the decorators `@classmethod` and `@staticmethod`, both of which are outside the purview of this book!

Here is our new `Date` class after making these changes:

Date3.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def setDate(self,d,m,y):
9.           if Date.isValid(d,m,y):
10.              self.day,self.month,self.year = d,m,y
11.          else:
12.              print("Invalid date!")
13.
14.      def getDay(self): return self.day
```

```
15.      def getMonth(self): return self.month
16.      def getYear(self): return self.year
17.
18.      def print(self):
19.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
20.
21.      def isValid(d,m,y):
22.          Date.daysInMonth[2]=28
23.          if y<1 or y>9999: return False
24.          if Date.isLeap(y): Date.daysInMonth[2]=29
25.          if m<1 or m>12: return False
26.          if d<1 or d>Date.daysInMonth[m]: return False
27.          return True
28.
29.      def isLeap(y): return y%4==0 and (not y%100==0 or y
%400==0)
30.
31.      def addDays(self,days):
32.          d,m,y = self.day,self.month,self.year
33.          Date.daysInMonth[2]=28
34.          if Date.isLeap(y): Date.daysInMonth[2]=29
35.
36.          for i in range(days):
37.              d=d+1
38.              if d>Date.daysInMonth[m]:
39.                  d=1
40.                  m=m+1
41.                  if m>12:
42.                      m=1
43.                      y=y+1
44.                      if Date.isLeap(y):
Date.daysInMonth[2]=29
45.                      else: Date.daysInMonth[2]=28
46.          result = Date()
47.          result.setDate(d,m,y)
48.          return result
49.
50.  d1 = Date()
51.  d1.setDate(1,2,2000)
52.  d2 = d1.addDays(100)
53.  d2.print()
```

Output:

```
11-5-2000
```

Observation:

1. This program is based on `Date2.py`. We have changed the instance methods `isValid()` and `isLeap()` to class functions by eliminating `self`.
2. A result of this change is that all calls to `isValid()` and `isLeap()` will now require `Date` instead of a `Date` instance.

12.4.5 Instance Methods as special Class Functions

Having understood class functions, we can view methods from a new perspective now: methods are class functions that accept a reference to the invoking object as the first argument! The following code snippet will illustrate this:

```
>>> class A:
...     def f(self):
...         print("Hello")
...
>>> a=A()
>>> a.f()
Hello
>>> A.f(a)
Hello
```

Observation:

1. We have defined a class `A` with a method `f` that prints "`Hello`" when invoked. We have an instance of this class whose reference is stored in `a`.
2. `a.f()` is a call to the method `f` of class `A` using `a` as the invoking object.
3. `A.f(a)` is a call to the class function `f` of the class `A`, passing `a` as an argument that is received as `self` in `f`.

Thus, methods are special class functions!

12.4.6 Public, Private and Protected Members

OOP generally talks about private, public and protected members of a class with this interpretation:

1. **Public** members are accessible everywhere where the class/object is accessible.
2. **Protected** members are accessible within the class that defines it as well as in all subclasses (classes that inherit it).
3. **Private** members are accessible only within the class and nowhere else.

Python does not directly support these concepts and all members of a class are inherently public. Python does support a convention however, that might help support this but does not enforce it. Since all class members are anyway public, we will only concentrate on protected and private members in this section.

12.4.6.1 Protected Members

As mentioned earlier, protected members are accessible within the class in which they are defined as well as in all it's subclasses.

Any member that starts with a single underscore indicates that it should be treated as a protected member. As already emphasized, this is only a *convention* and is not enforced by Python, but professionals treat this convention as a rule.

Syntax:

```
_member
```

Since we have not dealt with inheritance yet, protected members will be demonstrated in section 12.6.2.

12.4.6.2 Private Members

As mentioned earlier, private members are accessible only within the class in which they are defined and are not accessible anywhere else.

Any member that starts with minimum 2 underscores and ends with maximum 1 underscore indicates that it should be treated as a private member. This is achieved in Python typically by *name mangling*:

Private members of the form `__member_` get replaced by `_classname__member`. In other words, a member that starts with at least 2 underscores and ends with at most 1 underscore gets replaced to an underscore, followed by the name of the class to which the member belongs, followed by 2 underscores, followed by the member name. This prevents direct access to the member by it's name, but still does not prevent access via it's mangled name! Furthermore, this name mangling procedure may change in the future and hence programmers are advised not to use the mangled

name at any time to access a private member *illegally*!

Syntax:

```
__member
```

Here is a code snippet to demonstrate private members:

```
>>> class A:
...     def set(self,x,y):
...             self.x = x
...             self.__y = y
...     def print(self):
...             print("{},{}".format(self.x,self.__y))
...
>>> a=A()
>>> a.set(2,3)
>>> a.print()
2,3
>>> a.x
2
>>> a.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'y'
>>> a.__y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__y'
>>> a._A__y
3
```

Observation:

1. We have defined a class `A` with a setter called `set` that assigns the parameters `x` and `y` to the members `x` and `y` (called `__y`) respectively. The member `x` is considered to be public whereas the member `__y` is considered to be private.
2. The `print()` method displays the values of members `x` and `__y`, both of which are accessible because `print()` is a method of the same class.
3. From outside the class, only `x` is accessible and `__y` is not.
4. The private member `__y` is accessible outside the class with the name `_A__y`.

In our `Date` example, we can convert all class variables and instance variables to private! In general, it is always better to convert all class variables and instance variables to private to prevent accidental changes from outsiders and thus implement data hiding. Therefore, the instance variables `day`, `month` and `year` need to be renamed as `__day`, `__month` and `__year` respectively. Also the class variable `daysInMonth` will be replaced by `__daysInMonth`.

Date4.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       __daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def setDate(self,d,m,y):
9.           if Date.isValid(d,m,y):
10.              self.__day,self.__month,self.__year = d,m,y
11.          else:
12.              print("Invalid date!")
13.
14.      def getDay(self): return self.__day
15.      def getMonth(self): return self.__month
16.      def getYear(self): return self.__year
17.
18.      def print(self):
19.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
20.
21.      def isValid(d,m,y):
22.          Date.__daysInMonth[2]=28
23.          if y<1 or y>9999: return False
24.          if Date.isLeap(y): Date.__daysInMonth[2]=29
25.          if m<1 or m>12: return False
26.          if d<1 or d>Date.__daysInMonth[m]: return False
27.          return True
28.
29.      def isLeap(y): return y%4==0 and (not y%100==0 or y
%400==0)
30.
31.      def addDays(self,days):
32.          d,m,y = self.__day,self.__month,self.__year
33.          Date.__daysInMonth[2]=28
34.          if Date.isLeap(y): Date.__daysInMonth[2]=29
35.
36.          for i in range(days):
```

```
37.              d=d+1
38.              if d>Date.__daysInMonth[m]:
39.                  d=1
40.                  m=m+1
41.                  if m>12:
42.                      m=1
43.                      y=y+1
44.                      if Date.isLeap(y):
Date.__daysInMonth[2]=29
45.                      else: Date.__daysInMonth[2]=28
46.          result = Date()
47.          result.setDate(d,m,y)
48.          return result
49.
50.  d1 = Date()
51.  d1.setDate(1,2,2000)
52.  d2 = d1.addDays(100)
53.  d2.print()
```

Output:

```
11-5-2000
```

Observation:

1. This program is based on `Date3.py`. The instance variables (`day`, `month` and `year`) and class variables (`daysInMonth`) have been made private by prefixing them with __.
2. The instance and class variables can no longer be directly accessed outside the class.
3. Even class functions and instance methods can be made private by prefixing them with __. Such functions/methods can then be used internally by the class and is not directly callable from outside the class. In our `Date` example, we can consider making the class functions `isValid` and `isLeap` private if we feel that the outside world will have no interest in these.
4. Making class variables and instance variables private helps implement *data hiding*. Making class functions and instance methods private helps implement *data abstraction*.

12.5 Constructors and Destructors

In OOP, a **constructor** is a member function of a class that is automatically invoked when an instance of that class is created in order to initialize the object to a valid state. A **destructor** is a member function of a class that is automatically invoked when an instance is destroyed in order to perform any clean-up required.

Python does not have exact implementations for these, but does provide something very similar. This section focuses on Python's version of constructors and destructors.

12.5.1 Constructors

A constructor in Python is an instance method that is automatically invoked when an instance is created and permits the programmer to perform any initialization required to ensure that the instance is in a valid state.

A constructor is identified in Python by it's special name: `__init__`. Note that the leading underscores do not make this instance method private as the method name does not end with at most 1 underscore – it in fact ends with 2 underscores!

The following code snippet demonstrates how constructors can be designed and how they are automatically invoked when objects are instantiated:

```
>>> class A:
...     def __init__(self):
...             print("Constructor called!")
...
>>> a=A()
Constructor called!
>>> b=A()
Constructor called!
```

Observation:

1. The constructor is always called `__init__()` and being an instance method it will always receive `self` as the first parameter.
2. The constructor can be designed to receive any number of parameters in addition to `self`, but since Python does not support function overloading, only 1 constructor can exist in a class.
3. It is a good programming practice to use the constructor to create all instance variables that an object requires – all initialized to meaningful values.
4. In classes without explicitly defined constructors, one can imagine Python adding a dummy constructor that does nothing, i.e. an empty constructor.

In our `Date` example, we can add a constructor that permits the construction of a `Date` object by specifying the `day`, `month` and `year`. We need not worry about the implementation as the `setDate()` method can be used for this purpose. Since the objective of the constructor is to ensure that the object is in a valid state, we will ensure this even if the given values of `day`, `month` and `year` is not valid.

Date5.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class
4.
5.   class Date:
6.       __daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def __init__(self,d,m,y):
9.           self.setDate(1,1,1970)
10.          self.setDate(d,m,y)
11.
12.      def setDate(self,d,m,y):
13.          if Date.isValid(d,m,y):
14.              self.__day,self.__month,self.__year = d,m,y
15.          else:
16.              print("Invalid date!")
17.
18.      def getDay(self): return self.__day
19.      def getMonth(self): return self.__month
20.      def getYear(self): return self.__year
21.
22.      def print(self):
23.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
24.
25.      def isValid(d,m,y):
26.          Date.__daysInMonth[2]=28
27.          if y<1 or y>9999: return False
28.          if Date.isLeap(y): Date.__daysInMonth[2]=29
29.          if m<1 or m>12: return False
30.          if d<1 or d>Date.__daysInMonth[m]: return False
31.          return True
32.
33.      def isLeap(y): return y%4==0 and (not y%100==0 or y
%400==0)
34.
35.      def addDays(self,days):
36.          d,m,y = self.__day,self.__month,self.__year
37.          Date.__daysInMonth[2]=28
```

```
38.          if Date.isLeap(y): Date.__daysInMonth[2]=29
39.
40.          for i in range(days):
41.              d=d+1
42.              if d>Date.__daysInMonth[m]:
43.                  d=1
44.                  m=m+1
45.                  if m>12:
46.                      m=1
47.                      y=y+1
48.                      if Date.isLeap(y):
Date.__daysInMonth[2]=29
49.                      else: Date.__daysInMonth[2]=28
50.          result = Date()
51.          result.setDate(d,m,y)
52.          return result
53.
54.  d1 = Date(1,2,2000)
55.  d1.print()
56.  d2 = Date(90,90,90)
57.  d2.print()
```

Output:

```
1-2-2000
Invalid date!
1-1-1970
```

Observation:

1. This program is based on `Date4.py`. We have introduced a constructor in lines 8-10.
2. The constructor receives the `day`, `month` and `year` and uses the `setDate()` method to perform the necessary validation and assignment. It is a good practice to reuse functions as it reduces our work, makes the code more robust and also makes it easy to change the program later on if required.
3. Since our current implementation of `setDate()` merely prints an error message and continues if the given values of `day`, `month` and `year` do not represent a valid date, our constructor first sets a valid date (1-1-1970 is chosen primarily because it is a valid date, and more so because it is an important reference date in computers) and then only proceeds to call `setDate()` to change the date if the given date is valid. A better implementation will use exception handling, covered in section 13.

12.5.2 Destructors

A destructor in Python is an instance method that is automatically invoked when an object is going to be destroyed and eliminated and permits the programmer to perform any desired clean-up.

A destructor is identified in Python by it's special name: `__del__`.

The following code snippet demonstrates the working of destructors in Python:

```
>>> class A:
...     def __del__(self):
...             print("Destructor called!")
...
>>> a=A()
>>> del a
Destructor called!
>>> b=A()
>>> del b
Destructor called!
```

Observation:

1. The destructor is an instance method does not receive any parameters apart from `self`.
2. Like constructors, one class can have only one destructor, and the absence of a user-defined destructor results in a dummy destructor supplied by Python that is empty.
3. While in this example the destructor merely prints a message, destructors can do many meaningful operations when an object is going to be eliminated. If no such requirement exists, then a destructor need not be defined in that class.
4. The `del` built-in function is used to delete a variable, and can also end up deleting objects in memory.

Do recollect from section 2.6 however that there is a difference between objects/instances and references and that variables in Python only hold references! Deleting a variable using `del` not only deletes that variable, but also decrements the reference count of the referenced object by 1, and only when this reference count reaches 0 (implying that the object has no active references in the program), the object is deleted. This is demonstrated below:

```
>>> class A:
...     def __del__(self):
...             print("Destructor called!")
...
>>> a=A()
>>> b=a
>>> del a
>>> del b
Destructor called!
```

Observation:

1. We create an object of class `A` and store it's reference in the variable `a`. The reference count of this object is 1 (only 1 variable – `a` – is referring to the object)
2. This reference is copied from `a` to `b`, ending up with 2 references to the object. Therefore, the reference count of the object becomes 2.
3. When we delete the variable `a`, we also decrement the reference count of the object referred to by `a` by 1. Therefore, the reference count of the object decrements to 1. Since it has not reached 0, the object continues to exist unaffected by the deletion of the variable `a`.
4. When we delete the variable `b`, we also decrement the reference count of the object referred to by `b` by 1. The reference count now reaches 0. This is when the object will be destroyed, and just prior to that the destructor gets automatically called.

Since our `Date` class does not really require a destructor, we will not attempt to demonstrate the addition of the same.

12.6 Inheritance

Inheritance is one of the most important concepts in OOP and has these advantages:

1. It helps **compartmentalise** the code, thereby helping the programmer organise the code better.
2. It allows **reusability** of code by allowing a new class to completely obtain the functionality of other existing classes and add more functionality of it's own.
3. It permits **extensibility** of code, wherein new code is added without having to modify existing code and classes.

While the different structural forms of inheritance have been covered in section 12.1.7, we will basically examine 2 forms that make all other forms possible:

1. **Simple inheritance**, wherein one class inherits from another
2. **Multiple inheritance**, wherein one class inherits from multiple classes

One point to be kept in mind as far as inheritance is concerned is that the derived class contains *all* features of the base class. We can imagine that each instance of the derived class also contains an instance of the base class. The *private* features of the base class are also present in the derived class, but are not directly accessible.

12.6.1 Simple Inheritance

In simple inheritance, one class inherits from another existing class. The class that inherits is called the *derived* class or *subclass* while the class that is inherited from is called the *base* class or *super* class. Inheritance is therefore also termed *subclassing*.

The syntax of subclassing is shown below:

```
class derived_class(base_class):
    class_definition
```

Here is a small code snippet to demonstrate simple inheritance with class `B` deriving from class `A`:

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> a=A()
>>> b=B()
>>> type(a)
<class '__main__.A'>
>>> type(b)
<class '__main__.B'>
```

Observation:

1. The class definitions are empty, but as Python requires at least 1 line in the definition, we have used the `pass` statement.
2. The class `B` derives from class `A`: `class B(A)`

12.6.2 Private, Public and Protected Revisited

Section 12.4.6 had introduced the convention for *public*, *private* and *protected* access in a class. We had not seen protected access in action as inheritance was not covered then. Now that we have an idea of inheritance, let us revisit these and see them in action in an example:

inheritance1.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Public, Private and Protected Method access
5.
6.   class A:
7.    def __f1(self): print("A.f1")
8.    def f2(self): print("A.f2")
9.    def _f3(self): print("A.f3")
10.
11.  class B(A):
12.   def __g1(self): print("B.g1")
13.   def g2(self): print("B.g2")
14.   def _g3(self): print("B.g3")
15.
16.  b = B()
```

Observation:

1. The above program does not produce any output. It has been given only for us to analyse the program and draw conclusions. Therefore, there is no point in running this program.
2. Class `A` is the *base* class and class `B` is the *derived* class. Class `A` contains 3 methods - `__f1`, `f2` and `_f3` – which are *private*, *public* and *protected* respectively. Class `B` also contains 3 methods - `__g1`, `g2` and `_g3` – which are *private*, *public* and *protected* respectively.
3. Recollect that *public* members are accessible both inside as well as outside the class, *private* members are accessible only inside the class and *protected* members are accessible only inside the class and inside it's subclasses. Also recollect that the interpreter may not prohibit direct access of protected members from outside the class.
4. Line 16 creates an instance of the derived class `B` called `b`. We will now analyse which methods can be invoked using this instance `b`.
5. `b.g2()` is the most obvious valid candidate – any public method of a class is accessible from even outside the class.

6. `b.f2()` is the next obvious valid candidate – public methods of the base class and inherited as public methods of the derived class and public methods of the derived class are accessible even outside the class.
7. `b.__g1()` is an invalid candidate – private methods of a class are not accessible outside the class. We of course can invoke `b.f2()` which in turn can call `self.__f1()`.
8. `b._g3()` is supposed to be an invalid candidate – protected members of a class are not accessible outside the class, but the interpreter may not specifically prohibit it's access. We will only attempt to call `_g3()` through some other route (like through `g2()` for instance).
9. `b.__f1()` is an invalid candidate – private members of a class are accessible only within that class and are inaccessible even in it's subclasses. We can follow other routes though, like calling `__f1()` from `f2()` or from `g2()`.
10. `b._f3()` is supposed to be an invalid candidate – protected members of a class are accessible within subclasses, but not outside of these subclasses. `B` being a subclass of `A` does have access to `_f3()`, but we are not supposed to access them from outside `B`.

12.6.3 Function Overriding

The previous section illustrated which functions are accessible within derived classes and which are accessible even outside derived classes in inheritance. A special case arises however when we have functions with the same name in both the base as well as derived classes – this is called **function overriding**, wherein a derived class function is offered as a replacement for a base class function. This section concentrates on this concept.

When there are methods with the same name in a base class as well as it's derived class and we have a reference to an instance of the derived class, using which we invoke the function, the derived class function gets called. This is illustrated below:

Inheritance2.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A.f")
9.
10.  class B(A):
11.      def f(self):
12.          print("B.f")
13.
14.  b = B()
15.  b.f()
```

Output:

```
B.f
```

What if we want `B`'s `f` to also invoke `A`'s `f` as part of it's functionality? We can use the class function syntax to invoke the method (as illustrated in section 12.4.5), passing `self` as the reference to the instance. Alternatively, we can use the `super()` built-in to access the super class object of the current object (identified by `self`):

Inheritance3.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A.f")
9.
10.  class B(A):
11.      def f(self):
12.          print("B.f")
13.          super().f()
14.
15.  b=B()
16.  b.f()
```

Output:

```
B.f
A.f
```

Observation:

1. The statement `A.f(self)` is very similar to the statement `self.f()`, with `self` referring to an instance of `A`. But in our case, `self` is referring to an instance of `B` and `self.f()` will result in a call to `B.f(self)` instead!
2. The statement `super().f()` is equivalent to the statement `self.f()` with `self` referring to an instance of `A` that is housed within an instance of `B` that is currently referred to by `self`!

What if we want to directly invoke `A`'s `f` without involving `B`'s `f` using an instance of `B`? Again, we can use the class function syntax as shown below:

Inheritance4.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A.f")
9.
10.  class B(A):
11.      def f(self):
12.          print("B.f")
13.          super.f()
14.
15.  b=B()
16.  A.f(b)
```

Output:

```
A.f
```

Observation:

1. This time, we do not use `b.f()`, which will always give preference to `B`'s `f`. We directly invoke `A.f` passing `b` as the argument.

12.6.4 Constructors and Destructors in Simple Inheritance

One of the important results of inheritance is that the derived class is dependent on the base class. While the base class can exist on it's own, the derived class is dependent on the base class for it's existence. Derived class instances are dependent on corresponding base class instances. Indeed we can say that every derived class instance contains a base instance within itself, without which it cannot exist.

Building on this, we can also say then that when a derived class is instantiated, the base class should get instantiated first and when a derived class instance is destroyed, it should be followed by the destruction of the base class instance too. In other words, the order of construction/creation should be from base class to derived class whereas the order of destruction should be from derived class to base class. This order honours our agreement that the base class/instance can survive without the derived class/instance but not vice-versa.

Unlike other programming languages like C++ where this is automatically enforced by the compiler, Python does not enforce it! It therefore becomes the responsibility of the programmer to ensure that this indeed takes place by explicitly calling suitable base class functions.

Let's begin with a simple code snippet to observe Python's default behaviour when it comes to constructors and destructors under inheritance:

Inheritance5.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Constructors and Destructors
5.
6.   class A:
7.       def __init__(self):
8.           print("A constructed")
9.       def __del__(self):
10.          print("A destroyed")
11.
12.  class B(A):
13.      def __init__(self):
14.          print("B constructed")
15.      def __del__(self):
16.          print("B destroyed")
```

```
17.
18.  b=B()
19.  del(b)
```

Output:

```
B constructed
B destroyed
```

Observation:

1. We observe that when we instantiate class `B`, class `B`'s constructor is invoked, but class `A`'s constructor is not automatically invoked.
2. Similarly, when we destroy the instance of class `B`, the destructor of class `B` is invoked, but the destructor of class `A` is not automatically invoked.

Ideally, we would want the constructors of both classes to be invoked during construction and the destructors to be similarly automatically invoked upon destruction. Since this is not automatically performed by Python, we need to change the snippet as follows to obtain the desired result:

`Inheritance6.py`

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Constructors and Destructors
5.
6.   class A:
7.       def __init__(self):
8.           print("A constructed")
9.       def __del__(self):
10.          print("A destroyed")
11.
12.  class B(A):
13.      def __init__(self):
14.          super().__init__()
15.          print("B constructed")
16.      def __del__(self):
17.          print("B destroyed")
18.          super().__del__()
19.
20.  b=B()
21.  del(b)
```

Output:

```
A constructed
B constructed
B destroyed
A destroyed
```

Observation:

1. The derived class constructors calls the base class constructor *before* executing any code within it. This is ideally how derived class constructors should be!
2. The derived class destructor calls the base class destructor *after* executing any code within it. This is ideally how derived class destructors should be!
3. While it may not be considered wrong in Python if such calls are not made, it is generally required. If you are unsure of whether or not these calls are required in a particular situation, you might want to add the calls anyway. In fact, you could make it a habit of adding these calls whenever you write classes involving inheritance and remove it if and only when you have a strong reason to do so.
4. When the base class constructor requires arguments, the arguments could be received by the derived class constructor and passed on to the base class constructor in the call. In such cases, the derived class constructor is free to accept more arguments than are required by the base class constructor, but the additional arguments would be meant to be used by the derived class constructor and should not be passed to the base class constructor.

Let us reimplement the `Date` example of section 12.5.1 using inheritance. Recollect that we were printing the `Date` using the `print()` method in UK format. What if we wanted US format? Would it be a good idea to modify the code to print in US format? No! That will cease the current functionality which might be required in other places and forces us to retest the entire code again with the changes in place. Would it be a good idea to write another `Date` class with a different name like `USDate`, copying the code from the `Date` class and then making the changes? No! Tomorrow if we end up changing the code of the `Date` class, we might have to make the same changes in the `USDate` class, making maintenance difficult, changes complicated and introducing a possibility of adding bugs. The best technique would be to derive `USDate` from the `Date` class, thereby automatically having all functionality of the `Date` class, and overriding the `print()` method providing a different implementation of the same. This is illustrated in the program below.

Date6.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Date class and USDate class using
inheritance
4.
5.   class Date:
6.       __daysInMonth=[0,31,28,31,30,31,30,31,31,30,31,30,31]
7.
8.       def __init__(self,d,m,y):
9.           self.setDate(1,1,1970)
10.          self.setDate(d,m,y)
11.
12.      def setDate(self,d,m,y):
13.          if Date.isValid(d,m,y):
14.              self.__day,self.__month,self.__year = d,m,y
15.          else:
16.              print("Invalid date!")
17.
18.      def getDay(self): return self.__day
19.      def getMonth(self): return self.__month
20.      def getYear(self): return self.__year
21.
22.      def print(self):
23.          print("{}-{}-
{}".format(self.getDay(),self.getMonth(),self.getYear()))
24.
25.      def isValid(d,m,y):
26.          Date.__daysInMonth[2]=28
27.          if y<1 or y>9999: return False
28.          if Date.isLeap(y): Date.__daysInMonth[2]=29
29.          if m<1 or m>12: return False
30.          if d<1 or d>Date.__daysInMonth[m]: return False
31.          return True
32.
33.      def isLeap(y): return y%4==0 and (not y%100==0 or y
%400==0)
34.
35.  class USDate(Date):
36.      def __init__(self,d,m,y):
37.          super().__init__(d,m,y);
38.
39.      def print(self):
40.          print("{}/{}/
{}".format(self.getMonth(),self.getDay(),self.getYear()))
41.
```

```
42.  d1 = Date(1,2,2000)
43.  d1.print()
44.  d2 = USDate(1,2,2000)
45.  d2.print()
```

Output:

```
1-2-2000
2/1/2000
```

Observation:

1. This program is based on `Date5.py` covered in section 12.5.1, but we have removed the `addDays()` method for simplicity. Lines 1-34 are from that program.
2. We have added a class called `USDate` in line 35, which derives from the `Date` class.
3. Line 36 provides a constructor for the derived class, which simply passes on the parameters to the constructor of the base class. Our derived class constructor has no other job to do. No destructor was present earlier and is still not required here.
4. Line 39 provides a new definition for the `print()` method, which displays the date in US format.

12.6.5 Multiple Inheritance

A class can derive from 2 or more base classes, resulting in multiple inheritance. The syntax for multiple inheritance is given below:

```
class className(baseClass1,baseClass2[,baseClass3...])
```

The above syntax shows at least 2 compulsory base classes as without that this would be called multiple inheritance in the first place! There is no limit to how many classes a class can extend from. The order of inheritance is an important property here and goes from left to right in the base class list. Thus, the first base class is `baseClass1`, the second base class is `baseClass2` and so on. The order of inheritance matters when we use built-ins like `super()`, and professionals always aim to ensure that constructor calls are always in the order of inheritance whereas destructor calls are always in the strict reverse order. These are demonstrated in the following sections.

The program given below demonstrates multiple inheritance:

Inheritance7.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Multiple Inheritance
5.
6.   class A:
7.       def fa(self):
8.           print("A called")
9.
10.  class B:
11.      def fb(self):
12.          print("B called")
13.
14.  class C(A,B):
15.      def fc(self):
16.          self.fa()
17.          self.fb()
18.          print("C called")
19.
20.  c=C()
21.  c.fc()
```

Output:

```
A called
B called
C called
```

Observation:

1. Class `A` is defined in line 6 and contains a method `fa` defined in line 7.
2. Class `B` is defined in line 10 and contains a method `fb` defined in line 11.
3. Class `C` is defined in line 14 and derives from class `A` and class `B`. The order of inheritance is `A` followed by `B`, though in this example we do not observe any result because of this order.
4. A method `fc` is defined in line 14 inside class C, which invokes methods `fa` and `fb` upon itself, available to it because of inheritance.

12.6.6 Function Overriding in Multiple Inheritance

Function Overriding was introduced in section 12.6.3 and this section will focus on function overriding in multiple inheritance. Let us start with our first program:

Inheritance8.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Multiple Inheritance - Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A")
9.
10.  class B:
11.      def f(self):
12.          print("B")
13.
14.  class C(A,B):
15.      def f(self):
16.          print("C")
17.
18.  c=C()
19.  c.f()
```

Output:

```
C
```

Observation:

1. Class `C`, defined in line 14, derives from both class `A` and class `B`.
2. All the classes (`A`, `B` and `C`) define their own respective copies of the method `f` (lines 7, 11 and 15).
3. When a call is made in line 19 using the statement `c.f()`, preference is given to the method `f` in class `C`.

The reason why we are revisiting function overriding with respect to multiple inheritance is that it gets interesting now when we use the `super()` function because the question arises: when there are 2 (or more) base classes, which base class is considered to be the super class? To answer this question is obtained by the order of inheritance, which was covered in section 12.6.5. To recall, the order of inheritance is from left to right, and in this example, the first base class of `C` is therefore `A`. The first preference is given to the method `f` in class `A`. Only if `A` (and it's super classes, in the order of inheritance) does not define method `f` will preference be given to class `B` (and it's super classes in the order of inheritance). This is demonstrated by the following program:

Inheritance9.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Multiple Inheritance - Function Overriding
5.
6.   class A:
7.       def f(self):
8.           print("A")
9.
10.  class B:
11.      def f(self):
12.          print("B")
13.
14.  class C(A,B):
15.      def f(self):
16.          super().f()
17.          print("C")
18.
19.  c=C()
20.  c.f()
```

Output:

```
A
C
```

What if we want to call the method `f` of class `B` from within class `C`? We use the class function syntax: `B.f(self)`.

12.6.7 Constructors and Destructors in Multiple Inheritance

Section 12.6.4 emphasized on the order of construction and destruction and explained how this can be done by the programmer. The following program puts it all together to illustrate the best way of writing programs in Python that employ multiple inheritance:

inheritance10.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Multiple Inheritance - Constructors and Destructors
5.
6.   class A:
7.       def __init__(self):
8.           print("A constructed")
9.       def __del__(self):
10.          print("A destroyed")
11.
12.  class B:
13.      def __init__(self):
14.          print("B constructed")
15.      def __del__(self):
16.          print("B destroyed")
17.
18.  class C(A,B):
19.      def __init__(self):
20.          A.__init__(self)
21.          B.__init__(self)
22.          print("C constructed")
23.      def __del__(self):
24.          print("C destroyed")
25.          B.__del__(self)
26.          A.__del__(self)
27.
28.  c=C()
29.  del(c)
```

Output:

```
A constructed
B constructed
C constructed
C destroyed
B destroyed
A destroyed
```

Observation:

1. The constructor of class `C` first passes control to the constructor of class `A`, then the constructor of class `B` and then continues it's execution.
2. The destructor of class `C` first executes itself and finally calls the destructor of class `B` followed by the destructor of class `A`.
3. As pointed out in section 12.6.4, this is done to honour the rule that base classes can exist without derived classes but not the other way around.
4. The order of inheritance is respected: class `A` is the first base class of class `C` and hence is created first but destroyed last.

12.7 Dynamic Polymorphism

A very important feature of OOP is *dynamic polymorphism* – the mechanism of deciding which function to invoke at runtime depending on the type of the invoking object.

Before actually understanding dynamic polymorphism, it is important to realize that *subclasses are perfectly substitutable for base classes*, meaning that in any situation where we require a reference to a base class instance, a reference to a derived class instance will do equally well. Let us illustrate that through a sample program:

Inheritance11.py

```
1.   #!/usr/bin/python
2.
3.   # Inheritance Demo:
4.   # Dynamic Polymorphism
5.
6.   class Animal:
7.       def __init__(self,name):
8.           self.name = name
9.
10.      def speak(self):
11.          pass
12.
13.  class Dog(Animal):
14.      def __init__(self):
15.          super().__init__("Dog")
16.
17.      def speak(self):
18.          print("Bow wow!")
19.
```

```
20.  class Cat(Animal):
21.      def __init__(self):
22.          super().__init__("Cat")
23.
24.      def speak(self):
25.          print("Meow!")
26.
27.
28.  def introduce(animal):
29.      print("Hi! This animal is called",animal.name)
30.      print("This animal says: ",end='')
31.      animal.speak()
32.
33.  animal = Dog()
34.  introduce(animal)
35.  animal = Cat()
36.  introduce(animal)
```

Output:

```
Hi! This animal is called Dog
This animal says: Bow wow!
Hi! This animal is called Cat
This animal says: Meow!
```

Observation:

1. Class `Animal` is defined in line 6. It contains a constructor and a method called `speak`.
2. The constructor (defined in line 7) receives the name of the animal and stores it in the attribute `name`.
3. The `speak` method (defined in line 10) does nothing and will be overridden by the derived classes suitably.
4. The `Dog` class (defined in line 13) derives from the `Animal` class. It's constructor receives nothing, but invokes the constructor of `Animal` passing "`Dog`" as the name of the animal. It's `speak` method ends up printing "`Bow wow!`"
5. The `Cat` class (defined in line 20) derives from the `Animal` class. It's constructor receives nothing, but invokes the constructor of `Animal` passing "`Cat`" as the name of the animal. It's `speak` method ends up printing "`Meow!`"

6. The `introduce` function (defined in line 28) accepts an animal, prints it's name and invokes it's `speak` method to make the animal “speak”. While the function is designed to receive an instance of the type `Animal`, it can also receive an instance of any derived class of `Animal` because of the law of substitutability introduced at the beginning of this section. If a `Dog` instance is passed, preference is given to the `speak` method of `Dog` and if a `Cat` instance is passed, preference is given to the `speak` method of `Cat`. If `Dog` or `Cat` class does not override the `speak` method of `Animal`, then the speak method of `Animal` gets invoked which does nothing.

12.7.1 Abstract Methods and Classes

The previous program shows 2 disadvantages of the way the way the `speak` method has been defined in `Animal`:

1. We are forced to provide an empty definition of the method, which can seem a little odd at best.
2. If any of the derived classes do not override the `speak` method, there is no output on invoking the `speak` method as the `speak` method of `Animal` gets invoked and it is empty. There was no way to force the derived class to override the `speak` method.

These 2 disadvantages can be eliminated. The class can be made an abstract class using the `abc` package and the method can be made an abstract method using `@abstractmethod`, both of which are outside the scope of this book.

12.8 Attribute Handling, Magic Functions and Operator Overloading

A very interesting feature in Python is that certain operations when performed on objects will result in certain standard functions being called. What is interesting in this is that it does not appear that such a call is taking place!

This same principle is also used to implement operator overloading in Python – the mechanism by which an operator plays the same logical role but the implementation depends on the operand types on which the operator operates.

While this section focuses on operator overloading, we start with certain standard functions that are called when some basic operations are performed on objects. After an understanding of this, we will proceed to add support for operators to our classes.

12.8.1 Attribute Handling

Section 12.4.1 introduced the fact that instances can have attributes of their own. We use the syntax `object.attribute` to access such attributes. There are functions available in Python to deal with attributes of instances.

12.8.1.1 The hasattr() Function

The `hasattr()` function tells whether a particular instance has a particular attribute or not.

Syntax:

```
hasattr(object,attribute)
```

This is a Boolean function that returns true only if the given instance contains the given attribute, as shown in the code snippet below:

```
>>> class A:
...     def __init__(self):
...             self.x=0
...
>>> a=A()
>>> hasattr(a,'x')
True
>>> hasattr(a,'y')
False
```

Observation:

1. We define a class `A` that contains a constructor which initializes an attribute `x` to `0` (thereby creating it). Since the constructor is invoked each time an instance is created, it is guaranteed that all instances of `A` will have the attribute `x` in them.
2. `hasattr(a,'x')` therefore returns `True` whereas `hasattr(a,'y')` returns `False` since no attribute `y` was created in the instance `a`.

12.8.1.2 The getattr() Function

The `getattr()` function returns the value of an attribute within an instance if it exists, returning a default value (if provided) if the attribute does not exist.

Syntax:

```
getattr(object,attribute[,default])
```

Note:

1. If the attribute `attribute` exists in the instance `object`, it's value is returned.
2. If the attribute `attribute` does not exist in the instance `object`, `default` is returned.
3. If the attribute `attribute` does not exist in the instance `object` and no `default` is provided, an `AttributeError` occurs.

Example:

```
>>> class A:
...     def __init__(self):
...             self.x=0
...
>>> a=A()
>>> getattr(a,'x',2)
0
>>> getattr(a,'y',2)
2
>>> getattr(a,'y')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'y'
```

Observation:

1. `getattr(a,'x',2)` returns the value of the attribute `x` in the instance `a`, which is `0`.
2. `getattr(a,'y',2)` returns the value `2` (the `default`) as there is no attribute `y` in the instance `a`.
3. `getattr(a,'y')` generates an `AttributeError` as there is no attribute `y` in the instance `a` and no `default` was provided either.

12.8.1.3 The setattr() Function

The `setattr()` function sets the value of an attribute in an instance. If the attribute already existed in the instance, it's value is overwritten and if it did not exist, it is created.

Syntax:

```
setattr(object,attribute,value)
```

Example:

```
>>> class A:
...     def __init__(self):
...             self.x=0
...
>>> a=A()
>>> setattr(a,'x',2)
>>> setattr(a,'y',3)
>>> a.x
2
>>> a.y
3
```

Observation:

1. An instance `a` is created with an attribute `x` having the value `0`.
2. `setattr(a,'x',2)` overwrites the value of the attribute `x` in the instance `a` to `2`.
3. `setattr(a,'y',3)` creates an attribute `y` in the instance `a` and assigns a value `3` to it.

12.8.1.4 The delattr() Function

Finally, to remove an existing attribute from an instance, the `delattr`() function can be used.

Syntax:

```
delattr(object,attribute)
```

Note:

1. If the attribute `attribute` exists in the instance `object`, it is removed from the instance.
2. If the attribute `attribute` does not exist in the instance `object`, an `AttributeError` is generated.

Example:

```
>>> class A:
...     def __init__(self):
...             self.x=0
...
>>> a=A()
>>> a.x
0
>>> delattr(a,'x')
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'x'
>>> delattr(a,'y')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: y
```

12.8.1.5 Standard Attributes

While the programmer can create and delete attributes within an instance, there are certain standard attributes that are always present within each class (and therefore within each instance of that class), which are listed in the table below:

Attribute	*Details*
`__name__`	The name of the class
`__doc__`	The documentation string of the class (see section 12.2)
`__bases__`	A tuple containing the base classes of this class, in the order of inheritance (see section 12.6.5)
`__module__`	The name of the module to which this class belongs (see section 15)
`__dict__`	A dictionary containing the namespace of this class

Table 24: Standard Attributes

These attributes are illustrated in the program below:

StandardAttributesDemo.py

```
1.   #!/usr/bin/python
2.
3.   # Standard Attributes Demo:
4.
5.   class A:
6.       pass
7.
8.   class B:
9.       pass
10.
11.  class C(A,B):
12.      "A class to demonstrate standard attributes"
13.
14.      x=10
15.
16.      def __init__(self):
17.          pass
18.
19.      def f(self):
20.          pass
21.
22.  print("__name__:",C.__name__)
23.  print("__doc__:",C.__doc__)
24.  print("__bases__:",C.__bases__)
25.  print("__module__:",C.__module__)
26.  print("__dict__:",C.__dict__)
```

Output:

```
__name__: C
__doc__: A class to demonstrate standard attributes
__bases__: (<class '__main__.A'>, <class '__main__.B'>)
__module__: __main__
__dict__: {'f': <function C.f at 0x7f92c18564d0>, '__module__':
'__main__', '__doc__': 'A class to demonstrate standard
attributes', '__init__': <function C.__init__ at
0x7f92c1856440>, 'x': 10}
```

Observation:

1. We have defined class `A` in line 5 and class `B` in line 8. Class `C` (defined in line 11) derives from class `A` and class `B`. Class `C` contains a constructor and a method `f`. It also contains a class variable `x`.
2. We observe that the `__name__` attribute correctly gives us the class name as `C`.
3. We observe that the `__doc__` attribute picks up the documentation string from the class `C`.
4. We observe that the `__bases__` attribute has identified class `A` and class `B`, both in the module `__main__`, to be the base classes.
5. We observe that the module within which class `C` is present is `__main__`.
6. Finally, the attribute `__dict__` is a dictionary containing standard attributes as well as user-defined ones. Specifically, we observe that the dictionary contains the method `f`, the constructor and the class variable `x`.

12.8.2 Magic Functions

We use the term "*Magic Functions*" to denote those functions/methods that are automatically invoked without us explicitly naming them! All of these magic functions have the speciality that they have the following syntax for their name:

```
__name__
```

12.8.2.1 Constructors and Destructors

Here are some examples of magic functions that we have already used:

1. Whenever we create an object, it's constructor (`__init__`) is automatically invoked!
2. Whenever we delete an object reference using `del`, it's destructor (`__del__`) is automatically invoked!

Observe in the examples above that the functions being called have names that both start and end with double underscores and that we never explicitly called them! Here is a proof:

```
>>> class A:
...     def __init__(self): print("Created")
...     def __del__(self): print("Destroyed")
...
>>> a=A()
Created
>>> del(a)
Destroyed
```

Observation:

1. When we created an instance of class `A` using the statement `a=A()`, we observe that the `__init__()` method was automatically invoked!
2. When we deleted the variable a using the statement `del(a)`, we observe that the `__del__` method was automatically invoked!

12.8.2.2 Stringification

Since many a times objects are instances of custom user-defined classes that the Python interpreter had no idea about before your program could start execution, we find it convenient to have a string representation for such objects, especially when we want to display objects to the user or log them to a file. We use the term "*stringification*" to describe the conversion of an object to a string, and we use the following syntax to perform stringification:

```
str(object)
```

Note:

1. This is the constructor of the string class (`str`) that we had discussed in section 2.3.4.1.
2. The above statement does not work directly for user-defined classes, but can be made possible using the concept explained below.

Any attempt to stringify an object will automatically call the magic method `__str__` of that object. If the object does not contain this method, it will use the version provided by the (nearest) super class, calling this function of the `object` class in the worst case. This is demonstrated in the example below:

```
>>> class A: pass
...
>>> a=A()
>>> str(a)
'<__main__.A object at 0x7f390c039d30>'
```

Observation:

1. We created an empty class `A` and created an instance of it, the reference to which was stored in the variable `a`.
2. We see that though the class `A` did not provide an implementation for the `__str__` method, there was no error and we continue to get a string version due to the implementation in the `object` class.

Let us add a method by name `__str__` in class `A` to return a string and observe the behaviour:

```
>>> class A:
...     def __str__(self):
...             return "Hi"
...
>>> a=A()
>>> str(a)
'Hi'
```

Observation:

1. This time, our class `A` contains an implementation for the `__str__` method, which is designed to return the string "`Hi`" when invoked.
2. We create an instance of class `A` and store it's reference in the variable `a`.
3. We observe that `str(a)` ends up returning the string "`Hi`", proving that `str(a)` ended up making the call `a.__str__()`.

Some of the sample programs in section 12.10 will make use of this to provide appropriate string representations of objects of user-defined classes.

12.8.3 Operator Overloading

Most of the basic operators can work even with objects of user-defined classes if the operation methodology is taught to the Python interpreter using the concept of *operator overloading*.

12.8.3.1 Overloading Basic Arithmetic Operators

To illustrate the concept, if we have an object `o` and an integer `i`, it is possible to do operations like `o + i` if we overload the operator `+` in the class to which the object `o` belongs.

Here is an example without operator overloading:

```
>>> class A: pass
...
>>> o=A()
>>> i=5
>>> o+i
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'A' and 'int'
```

Observation:

1. We have defined an empty class `A`.
2. The variable `o` refers to an instance of class `A` and the variable `i` refers to an instance of `int` class with value `5`.
3. Due to the fact that we have not overloaded the `+` operator in class `A`, we see that the operation `o+i` does not work.

To support overloading of the `+` operator, we need to add the method `__add__` in class `A` as follows:

```
>>> class A:
...     def __add__(self, x): pass
...
>>> o=A()
>>> i=5
>>> o+i
>>>
```

Observation:

1. This time we observe that `o+i` does not give any error. This is because `o+i` results in the following call: `o.__add__(i)`.
2. Our `__add__()` method did not do anything since this is a demonstration. For the same reason, the method also did not return anything.

Some programs in section 12.10 will demonstrate practical uses of overloading operators like these. The table below summarises the magic functions for overloading basic arithmetic operators:

Arithmetic Operator	*Magic Function*
+	__add__
-	__sub__
*	__mul__
//	__floordiv__
/	__truediv__
%	__mod__
divmod()	__divmod__
**, pow()	__pow__

Table 25: Magic Functions for Arithmetic Operators

While we know that it is now possible to evaluate `o+i` since it maps on the `o.__add__(i)`, the question now is what about evaluating expressions like `i+o`? Since `i` is a reference to the built-in `int` class, there is no way that class can define such an operation to work on an instance of a user-defined class, as illustrated below:

```
>>> class A:
...     def __add__(self, x): pass
...
>>> o=A()
>>> i=5
>>> i+a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

The solution is additional magic methods that work in the reverse order! Thus, `i+o` can be implemented using `o.__radd__(i)`, where `__radd__` is the "reverse addition" magic method, as illustrated below:

```
>>> class A:
...     def __radd__(self, x): pass
...
>>> o=A()
>>> i=5
>>> i+o
>>>
```

Observation:

1. We observe that `i+o` works without errors since it maps on to `o.__radd__(i)`.
2. The `__add__` and `__radd__` magic methods are related, but neither relies on the other and it is not mandatory to have both, though possible and in many cases recommended.
3. Our `__radd__()` method did not do anything since this is a demonstration. For the same reason, the method also did not return anything.

The table below lists the magic functions to perform reverse arithmetic operations:

Reverse Arithmetic Operator	*Magic Function*
+	__radd__
-	__rsub__
*	__rmul__
//	__rfloordiv__
/	__rtruediv__
%	__rmod__
divmod()	__rdivmod__
**, pow()	__rpow__

Table 26: Magic Functions for Reverse Arithmetic Operators

12.8.3.2 Overloading Unary Arithmetic Operators

Just as how section 12.8.3.1 introduced magic methods for basic binary arithmetic operators, the table below summarises the magic methods for basic unary arithmetic operators:

Unary Arithmetic Operator	*Magic Method*
`-`	`__neg__`
`+`	`__pos__`
`abs()`	`__abs__`

Table 27: Magic Methods for Unary Arithmetic Operators

NOTE:

Unlike the operators shown in section 12.8.3.1, *reverse unary arithmetic operators* don't exist since these operators, being unary, work on a single operand and that operand is the instance of the class itself!

12.8.3.3 Overloading Type Conversion Operators

When objects of our class are being converted to other primitive types, certain standard magic methods are invoked to facilitate the conversion, if defined. These magic methods are summarised in the table below:

Conversion	*Magic Method*
`int()`	`__int__`
`float()`	`__float__`
`bool()`	`__bool__`
`complex()`	`__complex__`
`str()`	`__str__`
`bytes()`	`__bytes__`
`index(), bin(), oct(), hex()`	`__index__`

Table 28: Magic Methods for Type and Base Conversion

Note:

1. The `__str__` magic method was already covered formally in section 12.8.2.2.
2. The `__index__` magic method is supposed to return the same value as `__int__`, and the presence of `__index__` should ideally also be accompanied by `__int__`, though the presence of `__int__` does not mandate implementing `__index__`!
3. The `__index__` magic method is used when converting the integer equivalent of an instance to other bases, as performed by the `bin()`, `oct()` and `hex()` functions. All these functions use the return value of `__index__` and perform base conversions themselves.

12.8.3.4 Overloading Comparison Operators

In order to compare two objects, the relational operators can be used provided suitable magic methods are implemented as summarised in the table below:

Comparison Operator	*Magic Method*
<	`__lt__`
>	`__gt__`
<=	`__le__`
>=	`__ge__`
==	`__eq__`
!=	`__ne__`

Table 29: Magic Methods for Comparison Operators

12.8.3.5 Overloading Bitwise Operators

When bitwise operators are used on objects, corresponding magic methods get invoked. The table below summarises the magic methods invoked when the first operand is an instance of our class:

Bitwise Operator	*Magic Method*
&	__and__
|	__or__
^	__xor__
<<	__lshift__
>>	__rshift__
~	__invert__

Table 30: Magic Methods for Bitwise Operators

If the first operand does not support the required bitwise operation, then the second operand's reverse magic method gets invoked as summarised in the table below:

Reverse Bitwise Operator	*Magic Method*
&	__rand__
|	__ror__
^	__rxor__
<<	__rlshift__
>>	__rrshift__

Table 31: Magic Methods for Reverse Bitwise Operators

Note:

1. If the second operand also does not support the reverse bitwise operation, an error is reported.
2. The ~ operator, being unary, does not have a corresponding reverse operator.

12.8.3.6 Overloading Assignment Operators

Note that in statements of the form `o1=o2`, where `o1` and `o2` are objects of any class, the assignment operator (=) cannot be overloaded as Python takes up the responsibility of assigning the reference from RHS to LHS. However, for the shorthand assignment operators, we do have corresponding magic methods that get invoked, as summarised in the table below:

Shorthand Assignment Operator	Magic Method
+=	__iadd__
-=	__isub__
*=	__imul__
//=	__ifloordiv__
/=	__itruediv__
%=	__imod__
**=	__ipow__
&=	__iand__
|=	__ior__
^=	__ixor__
<<=	__ilshift__
>>=	__irshift__

Table 32: Magic Methods for Shorthand Assignment Operators

NOTE:

These magic methods ideally should return a reference to a new instance of the class that represents the result of the operation.

12.9 Empty Classes

An empty class in Python can be useful that way for a variety of reasons. An empty class is defined as a class containing only 1 statement: `pass` as shown in the syntax below:

```
class className:
    pass
```

This section covers some of the valid reasons why we might encounter empty classes:

12.9.1 Empty Classes as Placeholders

Many a times while coding, the programmer wants to focus on one aspect of the code but does not want syntax errors due to absence of code in other parts of the program. A programmer who needs a particular class, but does not want to commit on it's attributes and behaviour at the moment can easily start with an empty class and fill in the class details at a later point of time. Even when the class details are not implemented, the class as such is usable and instances of that class can be created and used as well.

12.9.2 Empty Classes for Identification

Sometimes, the only reason we want a particular class is to differentiate the objects. This happens, for example, in exception handling where we want to process different exceptions in different manners. Such a class need not have any implementation at all and we can make do with an empty definition.

> **NOTE:**
>
> The example in this section involves exception handling which will be covered in section 13. Readers are advised to read this section only after completing section 13.

Here is an example illustrating operations on a stack implemented using a user-defined class `MyStack`, in which the `push()` method can raise `StackOverflowException` and the `pop()` method can raise `StackUnderflowException`:

MyStack.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of a fixed-length Stack
4.
5.   class MyStack:
6.
7.       def __init__(self,MAX_SIZE):
8.           self.MAX_SIZE = MAX_SIZE
9.           self.values=[]
10.
11.      def push(self,x):
12.          if len(self.values) == self.MAX_SIZE: raise
StackOverflowException()
13.          self.values.append(x)
14.
15.      def pop(self):
16.          if len(self.values) == 0: raise
StackUnderflowException()
```

```
17.          return self.values.pop()
18.
19.      def display(self):
20.          for i in self.values: print(i)
21.
22.  class StackOverflowException(Exception): pass
23.  class StackUnderflowException(Exception): pass
24.
25.  myStack = MyStack(3)
26.
27.  while True:
28.      try:
29.          print("1. Push")
30.          print("2. Pop")
31.          print("3. Display")
32.          print("4. Quit")
33.          choice = int(input("Enter choice:"))
34.
35.          if choice == 1: myStack.push(int(input("Enter
value to push:")))
36.          elif choice == 2: print("Popped:",myStack.pop())
37.          elif choice == 3: myStack.display()
38.          elif choice == 4: break
39.          else: print("Invalid Choice!")
40.      except StackOverflowException:
41.          print("Stack overflow!")
42.      except StackUnderflowException:
43.          print("Stack underflow!")
44.      except ValueError:
45.          print("Invalid number!")
46.
```

Output:

```
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:10
1. Push
```

```
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:20
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:40
Stack overflow!
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
10
20
30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Popped: 30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
10
20
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Popped: 20
1. Push
2. Pop
```

```
3. Display
4. Quit
Enter choice:2
Popped: 10
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Stack underflow!
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
1. Push
2. Pop
3. Display
4. Quit
Enter choice:4
```

Let us analyse the output in pieces:

```
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
1. Push
2. Pop
3. Display
4. Quit
```

When we start with an empty stack, we first have used option `3` to verify that we indeed have an empty stack. Let us now add items to the stack:

```
Enter choice:1
Enter value to push:10
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:20
1. Push
```

```
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:1
Enter value to push:40
Stack overflow!
1. Push
2. Pop
3. Display
4. Quit
```

As can be seen, we added `10`, `20` and `30`. But when we attempt to add `40`, we get a "`Stack overflow!`" message since the maximum size of our stack is `3`! We will now use option `3` to display the stack and observe the stack contents:

```
Enter choice:3
10
20
30
1. Push
2. Pop
3. Display
4. Quit
```

We see that `10`, `20` and `30` are in the stack, but not the rejected value `40`. Let us pop a single item and display the stack contents to verify that the popped item is indeed removed from the stack:

```
Enter choice:2
Popped: 30
1. Push
2. Pop
3. Display
4. Quit
Enter choice:3
10
20
1. Push
```

```
2. Pop
3. Display
4. Quit
```

The value `30` was popped out from the stack and is no longer in the stack. Let is pop out all items one by one:

```
Enter choice:2
Popped: 20
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Popped: 10
1. Push
2. Pop
3. Display
4. Quit
Enter choice:2
Stack underflow!
1. Push
2. Pop
3. Display
4. Quit
```

We get a "`Stack underflow!`" message when we try to pop out an item from an empty stack. Let us verify that the stack is indeed empty:

```
Enter choice:3
1. Push
2. Pop
3. Display
4. Quit
```

Finally, let us quit the program:

```
Enter choice:4
```

Observation:

1. We have define the `MyStack` class from line 5.
2. The constructor defined in line 7 accepts `MAX_SIZE` – the maximum size of the stack – and stores it in the instance variable `MAX_SIZE`. The constructor also creates an empty list to house the stack items and stores it in the instance variable `values`.
3. The `push()` method defined in line 11 accepts a value to be pushed on to the stack. It first checks whether the stack is already full, and if so raises `StackOverflowException` instead. Otherwise it appends the given value to the list of stack values.
4. The `pop()` method defined in line 15 is supposed to pop out an item from the stack and return it. First the method checks if the stack is empty, raising `StackUnderflowException` in that case, else it pops out an element from the list of stack values and returns the element.
5. The `display()` method defined in line 19 displays the contents of the stack by iterating through the list of stack values.
6. The exceptions being raised in lines 12 and 16 are instances of classes `StackOverflowException` and `StackUnderflowException`, defined in lines 22 and 23 respectively. Recall from section 13.6 that such classes need to derive from the `Exception` class. Since no other functionality is required, the classes are empty!
7. Lines 40 and 42 show why we needed to differentiate between these classes for identification of the exception.

12.9.3 Empty Classes as Base Classes

When we have a class hierarchy and feel that certain classes "belong" to the same group, we decide to inherit them from a common base class to establish this logical belongingness. In such cases, it is perfectly valid for the common base class to have no implementation, if the programmer does not see any need. Thus, empty classes are used as base classes from which many related classes can derive.

We can change the program `MyStack.py` discussed in the previous section and ensure that the classes `StackOverflowException` and `StackUnderflowException` both derive from the common class `StackException` for better logical clarity!

MyStack2.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of a fixed-length Stack
4.
5.   class MyStack:
6.
7.       def __init__(self,MAX_SIZE):
8.           self.MAX_SIZE = MAX_SIZE
9.           self.values=[]
10.
11.      def push(self,x):
12.          if len(self.values) == self.MAX_SIZE: raise
StackOverflowException()
13.          self.values.append(x)
14.
15.      def pop(self):
16.          if len(self.values) == 0: raise
StackUnderflowException()
17.          return self.values.pop()
18.
19.      def display(self):
20.          for i in self.values: print(i)
21.
22.  class StackException(Exception): pass
23.  class StackOverflowException(StackException): pass
24.  class StackUnderflowException(StackException): pass
25.
26.  myStack = MyStack(3)
27.
28.  while True:
29.      try:
30.          print("1. Push")
```

```
31.          print("2. Pop")
32.          print("3. Display")
33.          print("4. Quit")
34.          choice = int(input("Enter choice:"))
35.
36.          if choice == 1: myStack.push(int(input("Enter
value to push:")))
37.          elif choice == 2: print("Popped:",myStack.pop())
38.          elif choice == 3: myStack.display()
39.          elif choice == 4: break
40.          else: print("Invalid Choice!")
41.      except StackOverflowException:
42.          print("Stack overflow!")
43.      except StackUnderflowException:
44.          print("Stack underflow!")
45.      except NumberFormatException:
46.          print("Invalid number!")
```

Observation:

1. This program is identical to `MyStack.py`, with changes in lines 22-24.
2. The output of the program is not shown as it is identical to the output of `MyStack.py`.
3. We decided to derive `StackOverflowException` and `StackUnderflowException` from `StackException` as we feel these 2 exception classes are logically related. The relation is established using the common base class `StackException`.
4. Due to exception handling rules (section 13.6), we need to ensure that `StackException` extends `Exception`. But apart from this, we have no need for adding any other piece of code within class `StackException`.
5. We can now also catch both `StackOverflowException` and `StackUnderflowException` using `StackException` in the `except` clause, if needed!

12.9.4 Empty Classes as Data Types

Since instance variables are not defined within classes, it is possible to start with an empty class but add instance variables directly in the instances themselves as shown in the example below:

```
>>> class A:
...     pass
...
>>> a=A()
>>> a.x=10
>>> a.y=20
>>> a.x
10
```

In such cases, having an empty class does not mean that the instances are empty!

Note for C/C++ Programmers:

The equivalent of the `struct` data type of C is an empty class whose instances can have any field required.

12.10 Programs based on OOP

This section shows some sample implementation of classes and concepts covered in this section.

12.10.1 Implementation of Counter Class

A counter is a mechanism for keeping track of an integer count that be incremented, decremented, increased, decreased and displayed. Here is an implementation of such a counter.

Counter.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Counter Class
4.
5.   class Counter:
6.       def __init__(self, count=0): self.set(count)
7.
8.       def set(self, count): self._count = count
9.       def get(self): return self._count
10.      def reset(self): self.set(0)
11.
12.      def increment(self, count=1): self.set(self.get()
+count)
13.      def decrement(self, count=1): self.set(self.get()-
count)
14.
15.      def __add__(self, x): return Counter(self.get()+x)
16.      def __radd__(self, x): return self.__add__(x)
17.      def __sub__(self, x): return Counter(self.get()-x)
18.      def __rsub__(self, x): return self.__sub__(x)
19.      def __iadd__(self, x):
20.          self.increment(x)
21.          return self
22.      def __isub__(self, x):
23.          self.decrement(x)
24.          return self
25.
26.      def __str__(self): return str(self.get())
27.      def __int__(self): return self.get()
28.      def __index__(self): return self.__int__()
29.
30.  # Basic counter setting/getting/resetting
31.  c = Counter()
32.  print(c.get())
33.  c.set(9)
34.  print(c.get())
35.  c.reset()
36.  print(c.get())
37.
38.  # Basic counter increment and decrement
39.  c = Counter(100)
40.  print(c.get())
41.  c.increment()
42.  c.increment(5)
43.  print(c.get())
```

```
44.  c.decrement()
45.  c.decrement(3)
46.  print(c.get())
47.
48.  # Basic counter operators
49.  c = Counter()
50.  c = c + 2
51.  c = 3 + c
52.  c += 5
53.  print(c.get())
54.  c = Counter()
55.  c = c - 2
56.  c = 3 - c
57.  c -= 5
58.  print(c.get())
59.
60.  # Counter type conversions
61.  c = Counter(12)
62.  print(int(c))
63.  print(str(c))
64.  print(c)
```

Output:

```
0
9
0
100
106
102
10
-10
12
12
12
```

Observation:

1. The `Counter` class is defined in line 5. The constructor in line 6 can optionally receive a count and store it within an instance variable `_count`. If this value is not given, it will be assumed to be `0`. The instance variable is *protected*, making it clear that the external world is not supposed to meddle with it directly but instead use *public* methods to gain access to it. These methods are basically `get()`, `set()` and `reset()`.
2. The `set()` method in line 8 accepts a count and sets it within the instance

variable `_count`.

3. The `get()` method in line 9 returns the count associated with the invoking object, stored in the instance variable `_count`.
4. The `reset()` method in line 10 resets the counter value back to `0`.
5. We see that the constructor is calling the `set()` method in line 6. Similarly, the `reset()` method is calling the `set()` method in line 10. We prefer reuse of method this way, even though the efficiency drops, since it makes the code more reliable and maintainable. For example, if we decide to change the name of the instance variable `_count` to `count` or `__count` or any other name, the change will be localised to `set()` (and `get()` too), but will have no impact on the constructor and `reset()` methods! We will also see to it that we don't reference the instance variable `_count` in any of our methods, preferring to use the `set()` and `get()` methods instead to indirectly access it.
6. The `increment()` method in line 12 increments the count of the invoking object by the specified value (default `1`). Again, this is done by calling the `set()` method. The `decrement()` method in line 13 similarly decrements the count of the invoking object by the specified value (default `1`).
7. Basic operator overloading is implemented in lines 15-24. Again, the focus is on reusability of methods rather than re-implementing within each method.
8. The `__add__()` method in line 15 is supposed to handle cases of the form `c+x`, where `c` is a `Counter` object and `x` is an integer. It is supposed to return the result of the expression, which we know should be a `Counter` instance.
9. The `__radd__()` method in line 16 is supposed to handle cases of the form `x+c`, where `c` is a `Counter` object and `x` is an integer. It is supposed to be identical to `c+x` in behaviour.
10. The `__sub__()` method in line 17 and `__rsub__()` method in line 18 similarly handle the cases `c-x` and `x-c` respectively where `c` is a `Counter` object and `x` is an integer.
11. The `__iadd__` method in line 19 handles the case `c+=x`, where `c` is a `Counter` object and `x` is an integer. Unlike the `__add__()` method which returns the result, here the result has to be stored within the invoking object itself, and we prefer to return a reference to the invoking object to support cascading of operations (not demonstrated here).
12. The `__isub__()` method in line 20 similarly handles the case `c-=x`, where `c` is a `Counter` object and `x` is an integer.
13. The methods in lines 26-28 handle conversion to other types. The `__str__()` method in line 26 helps convert `Counter` objects to strings. The `__int__()` method in line 27 helps convert `Counter` objects to integers.

The `__index__()` method in line 28 helps convert `Counter` objects to other bases if required (not demonstrated here).

14. The implementation of `__index__()` is supposed to be the same as that of `__int__()`.
15. The implementation of `__str__()` makes it possible for us to directly print `Counter` objects in a `print()` call by saying `print(c)`, where `c` is a `Counter` object.

12.10.2 Implementation of Distance Class

Here is an implementation of Distance class that can help represent any distance in a variety of units, and also supports the addition and subtraction of distances. The supported units currently are meters (m), kilometers (km), miles (mi), yards (yd) and feet (ft), though any other unit can be very easily added with minimal modification!

Distance.py

```
1.   #!/usr/bin/python
2.
3.   # Implementation of Distance Class
4.
5.   class Distance:
6.       _factors = {"m":1, "km":1000, "mi":1609.34,
"yd":0.9144, "ft":0.3048}
7.       def __init__(self, distance=0, unit="m"):
8.           self.set(distance, unit)
9.
10.      def set(self, distance, unit="m"):
11.          self._distance = Distance._normalize(distance,
unit)
12.
13.      def get(self, unit="m"):
14.          return Distance._externalize(self._distance, unit)
15.
16.      def _normalize(distance, unit):
17.          if unit == "m": return distance
18.          return distance * Distance._factors[unit]
19.
20.      def _externalize(distance, unit):
21.          if unit == "m": return distance
22.          return distance/Distance._factors[unit]
23.
24.      def add(self, distance, unit="m"):
25.          self.set(self.get() +
Distance._normalize(distance, unit))
26.
```

```
27.      def sub(self, distance, unit="m"):
28.          self.set(self.get() -
Distance._normalize(distance, unit))
29.
30.      def __int__(self): return int(self._distance)
31.      def __float__(self): return self._distance
32.      def __str__(self): return str(self._distance)
33.
34.  # Distance getting/setting in various units
35.  d = Distance(1000) #1000m = 1km
36.  print(d.get()) #1000m
37.  print(d.get("km")) #1km
38.  d.set(2,"km") #2km = 2000m
39.  print(d.get()) #2000m
40.  print(d.get("km")) #2m
41.  print(Distance(12,"ft").get("ft")) #12ft = 12ft
42.
43.  # Distance addition and subtraction
44.  d = Distance(500) #500m
45.  d.add(0.6,"km") #0.6km = 600m, 500+600=1100m
46.  print(d.get()) #1100m
47.  d.sub(1000) #1100-1000 = 100m
48.  print(d.get("km")) #100m = 0.1km
49.
50.  # Conversion to int and str
51.  d = Distance(500)
52.  print(int(d)) #500m -> 500
53.  print(str(d)) #500m -> "500"
54.  print(d) #d -> str(d)
```

Output:

```
1000
1.0
2000
2.0
12.0
1100.0
0.1
500
500
500
```

Observation:

1. The `Distance` class is implemented in lines 5-31. The constructor defined in

line 7 is capable of accepting an optional distance (`0` by default) in any unit required (meters by default). The assignment is made using the `set()` method.

2. The `set()` method defined in line 10 is responsible for assigning the specified distance to the instance variable `_distance`, after converting it to a standard unit (meters) using the method `_normalize()` for conversion (explained later).
3. The `get()` method defined in line 13 returns the distance represented by the invoking object in the specified unit (meters by default), converting the units using the method `_externalize()` (explained later).
4. Units conversion is important here as we are dealing with various units. We have stuck to meters as being a standard unit, though the program will behave exactly the same irrespective of which unit is chosen to be the standard! Based on the standard chosen, the *class variable* `_factors` (line 6) is populated as a dictionary with the keys as unit abbreviations and values as the multiplication factor that is to be used to convert that unit to the standard unit. Thus, to convert kilometers (km) into meters (m), the multiplication factor is `1000` and to convert miles (mi) into meters (m), the multiplication factor is `1609.34`. Obviously, to convert meters into meters, the multiplication factor is `1`! More units can be added here as required! Note that we have made this a class variable because it is common to all instances of this class!
5. The `_normalize()` method defined in line 16 is a *class method* that helps convert distances from other units to the standard unit (meters). As a special case, this method does not bother to convert units if it is already in meters. This is merely an *optimization* and conversion of units in such a case is not wrong, only unnecessary! The unit conversion is done using the class variable `_factors`.
6. The `_externalize()` method defined in line 20 is a class method that helps convert distances from meters to other units. This is the opposite function of `_normalize()`!
7. The `add()` method defined in line 24 adds the specified distance in the specified units (meters by default) to the invoking object.
8. The `sub()` method defined in line 27 similarly subtracts the specified distance in the specified units (meters by default) from the invoking object.
9. The `__int__()` method defined in line 30 converts the distance in meters to an integer; the `__float__()` method defined in line 31 merely returns the distance in meters (already a `float`/`int`) and the `__str__()` method defined in line 32 converts the distance in meters to a string.