What is a class?

A class is a code template for creating objects. Objects have member variables and have behaviour associated with them. In python a class is created by the keyword `class`.

An object is created using the constructor of the class. This object will then be called the `instance` of the class. In Python we create instances in the following manner

```
Instance = class(arguments)
```

How to create a class

The simplest class can be created using the class keyword. For example, let's create a simple, empty class with no functionalities.

```
>>> class Snake:
...     pass
...
>>> snake = Snake()
>>> print(snake)
<__main__.Snake object at 0x7f315c573550>
```

Attributes and Methods in class:

A class by itself is of no use unless there is some functionality associated with it. Functionalities are defined by setting attributes, which act as containers for data and functions related to those attributes. Those functions are called methods.

Attributes:

You can define the following class with the name Snake. This class will have an attribute `name`.

```
>>> class Snake:
...     name = "python" # set an attribute `name` of the class
...
```

You can assign the class to a variable. This is called object instantiation. You will then be able to access the attributes that are present inside the class using the dot `.` operator. For example, in the Snake example, you can access the attribute `name` of the class `Snake`.

```
>>> # instantiate the class Snake and assign it to variable snake
>>> snake = Snake()
```

```
>>> # access the class attribute name inside the class Snake.
>>> print(snake.name)
python
```

Methods

Once there are attributes that “belong” to the class, you can define functions that will access the class attribute. These functions are called methods. When you define methods, you will need to always provide the first argument to the method with a self keyword.

For example, you can define a class `Snake`, which has one attribute `name` and one method `change_name`. The method change name will take in an argument `new_name` along with the keyword `self`.

```
>>> class Snake:
...     name = "python"
...
...     def change_name(self, new_name): # note that the first argument is
self
...         self.name = new_name # access the class attribute with the self
keyword
...
```

Now, you can instantiate this class `Snake` with a variable `snake` and then change the name with the method `change_name`.

```
>>> # instantiate the class
>>> snake = Snake()

>>> # print the current object name
>>> print(snake.name)
python

>>> # change the name using the change_name method
>>> snake.change_name("anaconda")
>>> print(snake.name)
anaconda
```

Instance attributes in python and the init method

You can also provide the values for the attributes at runtime. This is done by defining the attributes inside the init method. The following example illustrates this.

```
class Snake:

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

```
    def change_name(self, new_name):
        self.name = new_name
```

Now you can directly define separate attribute values for separate objects. For example,

```
>>> # two variables are instantiated
>>> python = Snake("python")
>>> anaconda = Snake("anaconda")

>>> # print the names of the two variables
>>> print(python.name)
python
>>> print(anaconda.name)
anaconda
```

Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Private Variables

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

The above example would work even if `MappingSubclass` were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

```
>>>
```

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
```

```
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>>
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function