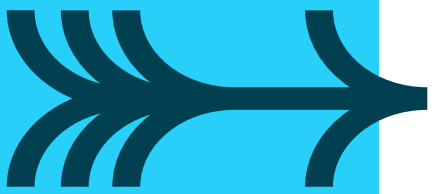
POWERING POTENTIAL

Python 3 Programming

Classes and Object Oriented Programming

CLASSES AND OOP



Contents

Object-orientation offers one of the best approaches to getting the most out of the Python language and libraries. This chapter presents and works through some of the key structuring concepts found in object modelling.

Using objects

Calling a class creates a new *instance object*

- Invokes the constructor

```
from account import Account

some_account = Account(1000.00)
some_account.deposit(550.23)
some_account.deposit(100)
some_account.withdraw(50)
print(some_account.getbalance())

another = Account(0)

print(Account.numCreated)
print("object another is class",
       another.__class__.__name__)
```

```
1600.23
2
object another is class Account
```

Just calling a class will call specific code to construct, or *instantiate* an object. Unsurprisingly, the function to construct an object is call a *constructor*, and it can be passed parameters in the usual way. In the Account example above, we are passing 1000.00 as a parameter to the constructor, presumably an opening balance. The constructor will return a reference to the object, which we can then call functions (methods) on. As we shall see, these functions are passed the object reference as their first parameter.

Object references are not normally printable, but we can provide our own *stringification* function, and others. This is part of special function overloading, which is related to operator overloading - a feature often seen as important in OO programming (although sometimes overdone).

The example shows a way of getting the name of the class that an object belongs to. This uses the attribute `__class__` which all objects have. In Python 3, you could also use the `type` built-in, but that gives output in a less useful format:

```
>>> print(type(another))
<class 'Account.Account'>
```

However, testing to see which class an object belongs to breaks

the principles of duck-typing - you should instead check to see if it quacks, i.e. use `hasattr` to see if it does what you need.

A class is not a type!

Don't ask what type an object is, only ask what the object can do

```
hasattr(object, name)
```

This is known as *duck-typing*

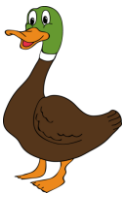
"*If it walks like a duck, swims like a duck, and quacks like a duck ...*"

```
if hasattr(x, '__str__'):
    val = str(x)
```

- In the example, we don't care what class **x** belongs to, only that it supports representation as a string

Based on the original concepts of object orientation

- Send/receive messages to/from an object
- *Signature-based* polymorphism



A foundation concept of object orientation (OO) is that messages are sent to objects requesting actions. We should not need to check the class of the object, only that it can carry out the action requested. That principle has been lost with many OO languages, particularly static ones. Even those that fully support polymorphism, the practice is often to test the class rather than the behaviour.
Python, and most dynamic languages, allows the programmer to check if behaviour exists. This is called *Duck Typing*: we don't care what kind of thing it is, if it quacks that's good enough. It could be some kind of duck mimic, who cares?

A little Python OO

- **A class is declared using `class`**
  - Membership is by *indentation*
- ***"class names use the CapWords convention"* – PEP008**
- **Methods are declared as functions within that class**
  - First argument passed is the object
  - The constructor is called `__init__`
  - The destructor is called `__del__`

→ Rarely required and unreliable

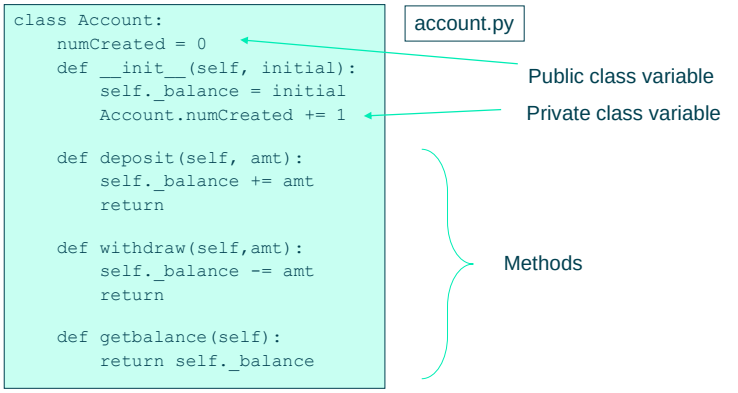
Classes are usually declared in a module

- File usually has same name as the class, with .py appended
- Simple example over…

Creating a class is very simple in Python, and an example is shown on the next slide. We shall be discussing the constructor in more detail later, but you can see that special function names begin and end with two underscores. There are many more than just the constructor and destructor.
The destructor, called when an object is destroyed, is rarely needed in Python because Python does its own memory management through reference counting. Remember that Python variables are actually references to objects, not objects themselves. So when a reference to an object drops out of scope, or gets reassigned, it does not necessarily destroy the object. The destructor does not get called until the reference count drops to zero, and even then Python does not guarantee to call it, even when the program shuts down! Do not rely on the destructor for committing transactions, closing network connections or flushing buffers - use exception handling and a finally block (discussed in another chapter) instead.

Defining classes

- **The class statement**
  - Defines a class object
  - → Public attributes are referenced by *Class.attribute*
  - Usually in a module with the same name as the class



Here is a simple example, which would be imported into a client program using:

```
from account import Account
```

Note the class variable `numCreated`, which is exported by default, since it's name does not begin with an underscore character.

Defining methods

- **Methods are functions defined within a class**
- **Conventions with underscores - reminder**
  - Names beginning with one underscore are private to a *module/class*
  - Names beginning with two underscores are private and mangled
  - Names surrounded by two underscores have a special meaning
  - Note: these do not guarantee privacy!
- **Object methods**
  - First argument passed to a method is the object

  → Usually called 'self', but can be anything
- **Class methods and attributes**
  - Defined within the class
  - Can be called on a class or object

The conventions for exporting names from modules apply with classes as well.
There are a number of *introspection* techniques which allow protected names to be seen, but the intention for privacy is clear. Class methods are not often required - it is usually better to implement a module function instead. It is not advisable to call a class method on an object - mainly because it looks confusing.

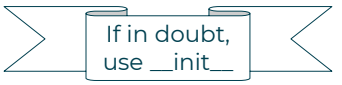
A function name prefixed by a single underscore indicates the name is meant for internal use only in a class. It's not really private like other languages but merely a convention for programmers to not access the names.

A name prefixed with a double leading underscore (dunders) has its name mangled by the Python interpreter in order to avoid naming conflicts in subclasses. Any identifier, such as __spam, will have its name changed to _classname__spam. This is helpful for letting subclasses override methods without breaking the parent classes methods.

Constructing an object

Python has two methods:

- __new__
  - → Called when an object is created
  - → First parameter is the class name
  - → Return the constructed object
- __init__
  - → Called when an object is initialised
  - → First parameter is the object
  - → An implicit return of the current object
- __new__ is called in preference



Which to use?

- Use __new__ only if constructing an object of a different class
- In most cases, use `__init__`

The constructor is a little confusing, having two possible functions. We rarely need a `__new__` function, usually we implement `__init__`, which will be called automatically. As can be seen from the slide, `__new__` is actually a class method, and is called before the object is created. We need `__new__` if we are going to create an immutable object (like a string or a tuple), which is not very often, or if we are going to use metaclasses. If `__new__` is provided, `__init__` is not called.

Python 2 note: in Python 2 `__new__` is only called for derived (*new-style*) classes. Base classes in Python 2 should inherit from `object` (more on inheritance later). All Python 3 classes are new-style classes.

Special methods

- **A mechanism for operator and special function overloading**
- Assists with duck-typing
- **Function names start and end with two underscores**

py3

`__bool__(self)`	Return True or False
`__del__(self)`	Called when an object is destroyed
`__format__(self, spec)`	`str.format` support
`__hash__(self)`	Return a suitable key for dictionary or set
`__init__(self, args)`	Initialise an object
`__len__(self)`	Implement the `len()` function
`__new__(class, args)`	Create an object
`__repr__(self)`	Return a python readable representation
`__str__(self)`	Return a human readable representation

Special methods are allied to overloaded operators only, here we are overloading Python built-in functionality. For example, **`__str__`** is called if an `str()` operation is done on our object, even implicitly such as in a `print` statement.
The Python 3 specific here is **`__bool__`**, which is called `__nonzero__` in Python 2.

Operator overload special methods

- **All operators may be overloaded**
- See the online documentation for a complete list

Return types vary

- Can return a `NotImplemented` object
- Examples:

`__add__`	+
`__sub__`	-
`__eq__`	==
`__ge__`	>=
`__lt__`	<
`__invert__`	~ (logical NOT)
`__getitem__(self,key)`	container element evaluation
`__setitem__(self, key, value)`	container element assignment

This is not a complete list of operator overload special methods, that is far too large to fit here - see the online documentation (search for `__add__(object method)`).
Compound (augmented) assignments, such as += have their own set of special methods, usually their names have a i prefix. For example, the special method for **+=** is **`__iadd__`**. Fortunately, if that is not supplied then Python looks for a **`__add__`** method and uses that instead, so we do not have to write versions of the same code (unlike C++).
However, methods overloading binary operators +, -, *, /, %,, **, <<, >>, &, ^, | are not automatically symmetrical, and require a method name prefixed '__r' to make them so. For example:

```
object + 1      calls __add__
1 + object      calls __radd__
```

Many of these methods are similar, and will often call an underlying method. For example, __eq__, __ne__, __lt__, __gt__, __le__, __ge__ can all be implemented as a single-line call to a generic comparison method. The comparison method will often be implemented as returning -1, 0, or +1 (where 0 means equality).

Special methods - example

```
class Date:

    def __init__(self, day=0, month=0, year=0):
        self._day   = day
        self._month = month
        self._year  = year

    def __str__(self):
        return "%02d/%02d/%d" % (
               self._day, self._month, self._year)

    def __add__(self, value):
        retn = Date(self._day, self._month, self._year)
        retn._day = retn._day + value
        retn._validate_date()
        return retn

today = Date(9, 10, 2015)
print(today)
tomorrow = today + 1
print(tomorrow)
```

Note private variable and method names starting with two underscores

```
09/10/2015
10/10/2015
```

We are using 'this' in one method and 'self' in others just to demonstrate that it can be done - don't do this at home! Choose one convention and stick with it consistently - if in doubt use '**self**', most Python programmers do and so does the Python documentation.

In the user code, the `print` statement will execute the **`__str__`** method, and the **`+=`** operator will execute **`__add__`**.

Notice in the **`__add__`** we are not altering the current object, we are altering (and returning) a copy. There are several ways we could copy the current object, we call the constructor (`Date()`) to create a new one from scratch.

However there might be additional attributes added by other methods. In this case `copy.deepcopy()` could be used instead. The `copy` module will do its best to copy the object, but sometimes that is not enough, in which case, the class can implement a `__deepcopy__` method, which will be called when `copy.deepcopy` is called on an object of that class.

The downside of not calling the constructor is that there might be a missing side-effect. For example, if the constructor kept a global (class variable) count of all the objects - `deepcopy()` would miss

that.
In general, the method shown on the slide is preferable, i.e. calling the constructor. It then follows that adding attributes to an object in later methods should be considered bad practice. If it can't be avoided, then a more complex constructor is required that optionally takes an object as an argument.
By the way, there are several date/time handling modules in the Python Standard Library.

Properties

Built-in property() creates an attribute

- property() has getter, setter, deleter, and docstring
- The appropriate method is called depending on the way the attribute is used

```
class Date:
    ...
    def mget(self):
        return self._day

    def mset(self, day):
        self._day = day

    mday = property(mget, mset)
```

Call the (default) getter method

```
day = today.mday
```

Call the setter method

```
today.mday = 6
```

Omitting the setter method means that the attribute is *read-only*

The `property()` built-in function defines get, set, delete, and docstring methods for a specific attribute. All parameters can be defaulted, following the usual rules. In the example, we have only used setter and getter methods.
When the attribute is used for read, for example on the right-hand side of an assignment, then the getter method (first parameter) is used. When used on the left-side of an assignment the setter is used.
In the example, if we used `del today.mday` then it would attempt to call a deleter method, but we have not provided one so that would fail with:

```
AttributeError: can't delete attribute
```

The docstring parameter is also missing in the example, but this time an attempt to use it will use the getter's docstring by default.

Properties and decorators

- **A decorator is a function name prefixed @**
  - The function will normally return another function
  - The decorator is followed by the function to be returned
- **Decorators are syntactic sugar, but commonly used**
  - Built-in property() is usually called using a decorator

```
class Date:
    ...
    @property
    def mday(self):
        return self._day

    @mday.setter
    def mday(self, day):
        self._day = day
```

Call the (default) getter method

```
day = today.mday
```

Call the setter method

```
today.mday = 6
```

The `property()` built-in function defines get, set, delete, and docstring methods for a specific attribute. It is rarely called in the conventional manner, but by using a *decorator*. Decorators only work with new-style objects (they might *appear* to work sometimes with old-style, but some aspects do not).
The `@` sign marks a *decorator expression*, and it can be applied to any function which takes function references as parameters, in this case the `property()` built-in. It is equivalent to calling the function (`property()`) and passing the supplied functions as parameters.
The example on the slide is equivalent to:

`property(`*first_mday*`, setter=`*second_mday*`)`

By setting `@property`, we define a default "`getter`" method which returns an attribute of the class. Notice that the attribute (`__day` in the example) is not defined elsewhere. We can also define a setter and deleter method by using the general syntax `@` + *getter_method_name* + `.setter` or `.deleter` (`docstring` is not supported using `property()` with a decorator).
Note that the names `setter`, `getter`, `deleter`, are not the keywords to `property()`, they are methods in the property class which can be used as decorators.

Decorator expressions are used elsewhere, although you may find the concept rather advanced. This is the most common, and there are others, including `@classmethod` (next slide), and `@context manager`. Modules such as Twisted, Turbogears and Django also use decorators.
You can create your own decorators, but this is outside the scope of this course.

Decorators - what's the point?

- **Decorators are part of Python function syntax**
  - Not specifically OO, but often used in OO contexts
  - Part of *metaprogramming*
- **One aim is to make code easier to read**
  - The decorator 'decorates' functionality

```
    def mget(self):
        return self._day
    ...
    mday = property(mget, mset)
```

Trailing property() call might be missed, or forgotten. When does it get executed?

```
    @property
    def mday(self):
        return self._day
    ...
```

Method is bound to the attribute name, and bound to @property.

The requirement for decorators comes from *metaprogramming*, which is programming capable of modifying the code itself at runtime.
Remember that decorators are not only used with properties, they can be used with any function that takes a function as its first (and subsequent) argument.
They are often used with built-ins, and they were originally added specifically to help with the syntax of creating class methods (see over).
Their justification is described in PEP 318 as an aid to readability. The problem with using built-ins like `property()` is they are not necessarily bound in code to the functions they are describing. We have to create new functions that have to be named differently to the attribute that they will be tied to. It might not be obvious that there is a logical link between these functions and the attributes.
The code we have shown so far is fairly simple, image trying to track properties with a large and complex class.

Class methods

There are several ways to achieve this

- Using a dummy class wrapper
- Using the classmethod built-in as a decorator (preferred)

→ The class method itself

```
    _count = 0
...
    @classmethod
    def get_count(cls):
        return Date._count
```

The class name is passed implicitly

→ The user of the class

```
from date import Date
...
counter = Date.get_count()
```

Class methods are not often required - it is often better to implement a module function instead. It is not advisable to call a class method on an object - mainly because it looks confusing.
If you try to define a function within a class without an argument, it is **unbound**, neither an object or a class method. In Python 2, this produced an error, but is now allowed.
A class method is considered **bound** to the class and gets the class name (called **cls** by convention) as a parameter. The **`@classmethod`** decorator is required to indicate this. An alternative is `@staticmethod`, which is more like a class method in Java or C++, but does not pass the class name.

Inheritance

Use attributes and methods from a parent class

- Important OO concept

→ Python supports multiple inheritance - not often needed

- Attributes and methods not supplied in the derived class will be inherited from the base class
- Common to derive our own classes from Python's own

→ Multithreading

→ Exceptions

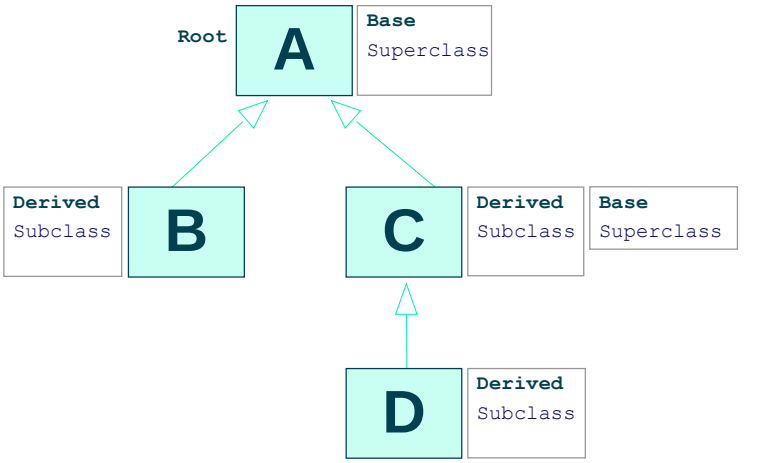
→ etc.

```
class DerivedClassName(base_classes):

    def __init__(self, arguments):
        base_class.__init__(self, arguments)

Other methods…
```

In object-oriented technology, inheritance provides an "is a" relationship between two classes. The derived class inherits all of the attributes and all of the operations of the base class. It can also add its own data members and member functions to provide more specialised characteristics.
Furthermore, the derived class can override a function in the base class, if it doesn't quite meet the specific requirements of the derived class. This is known as polymorphism.
Inheritance is a natural concept and we use it in our day-to-day lives. We learn at any early age to classify similar objects in terms of their fundamental characteristics. We speak of cars in general terms, without worrying too much about a particular type of car, such as four_wheel_drive_car or sports_car. As soon as we mention the word "car", the basic properties are understood because all cars have certain features in common; all cars have wheels, for example, and they all have brakes.
The same is true with inheritance in the world of object-oriented technology. Inheritance allows the class designer and implementer to group related classes together under a single umbrella, so that they can be considered collectively in general

terms or individually as specific classes.
By reusing tried and tested services of an existing class, applications can be developed more quickly and with a greater degree of confidence in the end product. Inheritance can result in code implosion, since the same base-class functions can be used in many derived classes. In Python, it is very common to derive our own classes from Python classes.

Inheritance terminology



There are a number of different methodologies available for expressing object-oriented designs, and many have an accompanying notation for representing object-oriented designs graphically. We shall adopt the simple notation shown to represent a hierarchy of derived classes, using arrows to indicate which is the base class. For example, class C is derived from class A and class D is derived from class C. So D inherits all the members of C, which in turn inherits all the members of A.

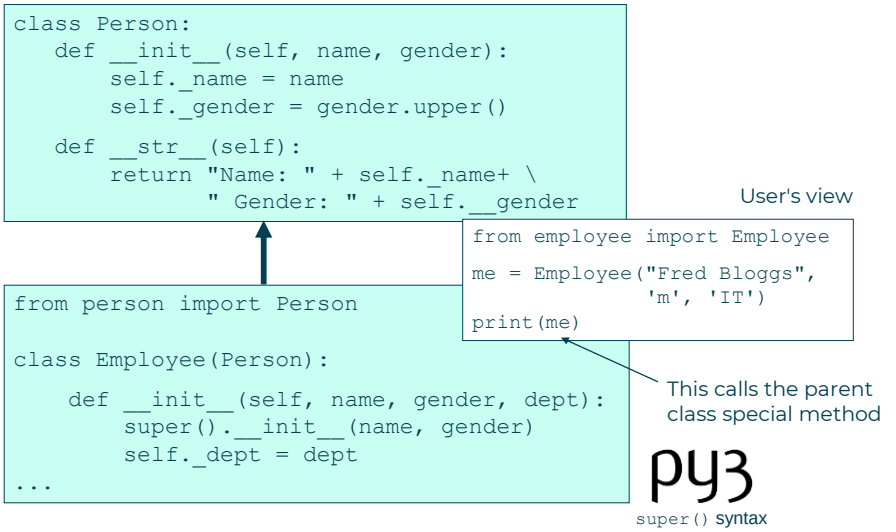
Inheritance scope

- **Attributes are either “public” or “private”**
  - No equivalent of “protected”
  - This includes methods as well as data items
  - Enforced by the leading two-underscores rule
  - Base and derived classes can share the same module

  → Can share attributes privately that are prefixed with a single underscore
- **Public attributes of the base class can be called on an object of the derived class**
  - Also applies to __special__ methods

When a derived-class object is used to call a method, and that method does not exist in the derived class, then a base-class method of that name is called (if one exists).
An interesting situation exists, if a base class and its derived class have the same method names. In this case, if a base-class method calls one of these duplicated method names and ‘self’ is a derived-class object, then the derived-class method is called, not the base-class one. This applies even if the duplicated method names have two leading underscores, i.e. are private.
The same problem exists with public data attributes, but not private ones.

Inheritance example



In this example, we have a base class called Person, and a derived class called Employee. The only thing which the Employee class does is provide its own constructor, although other methods would normally have been provided as well.
The Employee constructor (**`__init__`**) calls the base class constructor using **`super()`**, which returns a base class object. This is controversial, and many people specifically call the base class instead. The issue is with multiple inheritance: when we derive from several base classes, which one will **`super()`** return? So, instead of the call using **`super()`** we could have done:

```
Person.__init__(self, name, gender)
```

(notice that we have to explicitly pass `self`)
The syntax for **`super()`** changed at Python 3.

Some helper built-in functions

isinstance(*object*, *classinfo*)

- Returns True if *object* is of class *classinfo*

issubclass(*class*, *classinfo*)

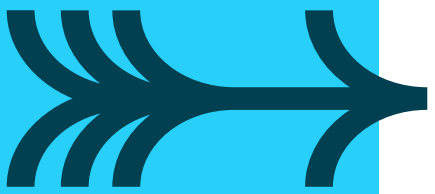
- Returns True if *class* is a derived class of *classinfo*

```
from employee import Employee
from person import Person

me = Employee("Fred Bloggs", 'm', 'IT')

if isinstance(me, Employee):
    print(me, "isa Employee!")

if isinstance(me, Person):
    print(me, "isa Person!")

if issubclass(Employee, Person):
    print("Employee is a subclass of Person")
```

All these conditions return True (based on the inheritance example)

In both cases, *classinfo* can be a tuple of classes. These functions could be considered to discourage duck-typing.

SUMMARY



- **Classes vs. objects**
- A class is a user defined data type
- An object is an instance of a class
- Objects have identity
- To achieve behavior we call an operation on an object
- The operations on an object are defined by its class
- **Encapsulation**
- Separates interface from implementation
- Publicly accessible operations
- Privately maintained state

Classes represent a convergence of the traditionally separate concepts of function and data. A class defines a new data type, providing it with a name, a set of operations expressed as member functions, and a representation expressed as member data. An object is a runtime instance of a class. There can be many objects of a single class, and many classes within a system.
The key benefit of objects is that they are self contained runtime components defined in terms of their external behaviour, collaborations and responsibilities, rather than in terms of their private representation. The combination of function and data within the same unit, and the protection of that data via an effective fire wall, is known as encapsulation.

Metaclasses and ABC

- **A metaclass is a class for creating other classes**
- The syntax for metaclasses changed at Python 3
- **Abstract Base Classes**
- Classes that cannot be directly instantiated
- Created metaclass ABCMeta and decorator abstractmethod

```
from abc import *

class Vehicle(metaclass = ABCMeta):
    @abstractmethod
    def getReg(self):
        pass

class Car(Vehicle):
    def getReg(self):
        print("Car isa Vehicle")
```

```
beepbeep = Car()
beepbeep.getReg()
```

```
Car isa Vehicle
```

```
NoGo = Vehicle()
```

```
Can't instantiate abstract class Vehicle...
```

ABCMeta is exported by the abc module. With meta-classes we might want to provide a `__new__` function, since the actual class used for the object creation could be different to the current class. See PEP 3119 for further details.