Object-Oriented Programming

Introduction

OOP is a style of programming, involving the creation of abstract models of real world entities.

Classes and Objects

There are many lights but they all have similar states and behaviours. The attributes that describe the light (state) and the actions that can be carried out on the light to change its state (behaviours) are encapsulated in a class.

The class acts as a blueprint for each instance of a light object.

Every instance has its own copy of the attributes, and when we call a method it acts only on the instance that we specify.

Encapsulation

A light has a state: on or off.

We can change that state with a switch. Encapsulation is the process of grouping state and actions together in a class.

Ideally the light's state is only changed through a method. This method will be written by the developer who wrote the class and fully understands it, while other developers won't know all the details.

The light class is a black box to other developers – they shouldn't have to understand what's going on in order to use it.

Only allowing variables to be changed by methods makes the code more robust.

This also allows the class to be changed (fixed and improved) without needing to change code that uses the class.

Inheritance

Say we've implemented a regular light and we want to now implement an LED light. Some extra attributes and methods are required, but a lot of the ones from the regular light apply to the LED light as well.

We want to reuse the functionality of the original light, overriding any existing methods that need to be specialised, and adding any new attributes and methods needed.

This is called Inheritance.

Composition

We can model complex objects which contain simpler objects as variables within them.

Rather than write a single class, we write a class (LightSystem) that is composed of the methods and actions required for the system but reuses the classes we have already written. We can use light and switch objects in the same way we use strings and integers.

Similar advantages to inheritance.

Advantages of OOP

- Helps us write cleaner code with fewer errors
- Code is easier to fix and fixes are invisible to developers
- Helps to reuse code

Python vs. Pure OOP Languages

Python allows you to write programs without using OOP, unlike languages like Java.

Python emphasises the freedom to use the language in any way. Encapsulation isn't enforced, for example.

In many ways Python has better OO support than Java, e.g. allowing the use of

Multiple Inheritance.

The Class

The class is the basic building block in OO design. Captures state with variables and behaviours.

Classes act as a factory for instantiating one or more objects.

An object exists in memory and has its own copy of variables to represent its state. These are called instance variables.

When we call a method, it acts on a particular object.

Our First Class

A class that manages information about employees.

In a file called person.py:

```
class Person:
    def __init__(self, name, job, pay):
        self._name = name
        self._job = job
        self._pay = pay
```

The `__init__` method is called the constructor – it's a function called when the class is instantiated.

We can then use the class:

```
cathal = Person('Cathal', 'dev', 55000)
laura = Person('Laura', 'manager', 70000)
print(cathal.name, cathal.pay)
print(laura.name, laura.pay)
```

When the first line here runs, a new instance is created in memory, using the class as a blueprint. The constructor is called to initialise the instance and add instance variables, based on the parameters passed through.

The second line follows the same procedure to create the second instance of the class.

Self

Self is a reference to the object instance. We don't need to pass this when we call the constructor, it's automatically pointed at the new object we create.

Packaging Our Class

Our code can be used by others in their code. They can include an entire module (python file) using the `import` statement. This identifies an external mile to be loaded and makes that name a reference to the module object after it is loaded:

```
import person

cathal = person.Person("Cathal", "dev", 55000)
print(cathal._name)
```

We have to use the module name before the class name, though, which is awkward.

A nicer way is to use the `from` statement:

```
from person import Person

cathal = Person("Cathal", "dev", 550000)
print(cathal._name)
```

Now we can use the Person class as if it was written in our code.

When we import the python module like that, though, all top-level code in the module is run. Two solutions:

- Have a separate test module in the package (not ideal)
- Have a test block in our module but control when it's run with a "usage mode flag" (better)

We can determine when a module is being run directly and when it is being imported. Every module has a `__name__` attribute.

- When we run a module directly, this attribute is set to `__main__`
- When the module is imported then `__name__` is set to the module's name

We can now have a separate test block:

```
class Person:
    [class definition]

if __name__ == '__main__':
    [test block]
```

This also allows us to import a module's functionality into a GUI or use the module as a standalone tool.

This is a good way for us to test our scripts in the labs.

Style

Alternatively, we can do this (suggested in the Google Style Guide):

```
class Person:
    [class definition]

    def main():
        [test block]

if __name__ == '__main__':
    main()
```

In general in this module we're gonna follow the Google Style Guide.

Encapsulation and Style

Python doesn't enforce instance variable privacy – a key component of encapsulation in other languages.

We use an underscore prefix for internal variables to indicate to users that the variable shouldn't be accessed directly, but should instead be accessed through methods.

Documenting Your Code

A doc string should be the first content in a module, class, or function, to describe what it is. See the Google Style Guide.

The doc string is contained by default in the `__doc__` attribute, and is supported/expected by many other tools, so it's useful to have it.

Outputting Objects

Say we want to print our objects. If we just use the print command the output isn't very useful for people.

Python objects have several special methods (like `__init__`), and we can use one (`__str__`) to return a representation of an object instance.

String Representation

`__str__` generates an informal representation. It has to have this name and it has to return a string.

We can write it ourselves to control the representation:

```
def __str__(self):

    description = ("%s %s %d" % (self._name, self._job,
self._pay))
    return description
```

Now in our main function, we can print an instance: `print(cathal)`

Methods

A method is a function that belongs to a class.

It is like a regular function except that:

- It is contained in a class
- Its first parameter is self
    - This causes it to act on the member variables of a particular instance of the class

Let's add a method to the Person class to allow modification of the pay attribute. We will pass the rise as a percentage of the person's salary:

```
def givePayRaise(self, percentage):

    if percentage < 0 or percentage > 100:
        print("%i is an illegal percentage" % (percentage))
    else:
        self._pay += self._pay // 100 * percentage
```

We have a check at the start to make sure the percentage isn't negative. We've also a check that the percentage isn't more than 100%.

Here's how we call it:

```
def main():

    cathal = Person('Cathal', 'dev', 70000)
    cathal.givePayRaise(10)
    print(cathal)
```

Supporting Encapsulation

We are already marking variables by starting their names with an underscore to indicate that users of our class should proceed with caution when editing variables like this.

It is advantageous to encourage developers to use methods we provide to change instance variable values:

- We can ensure values are changed in a controlled way.
    - E.g. we can add checks for values that don't make sense.
- We can ensure that third party code is not tightly bound to our instance variables. This gives us freedom to change our implementation without forcing changes to their code.

Remember that Python doesn't enforce encapsulation, unlike e.g. Java.

Getters and Setters

Version 1

Traditionally we set instance variables with "setters" and read them with "getters". These act on just one variable at a time, so we need one each for `_name`, `_job`, and `_pay`.

```
def getName(self):
    return self._name

def setName(self, name):

    if len(name) <= 0 or not name.isalpha():
        print("%s is not legal" % (name))
    else:
        self._name = name
```

`setName` performs a check before setting the name.

We use them as we would any other method:

```
def main():
    cathal = Person('Cathal', 'dev', 70000)
    cathal.setName('CathalNew')
    print(cathal.getName())
```

Version 2

Using properties, we can access the attributes through methods but using a syntax that's the same as accessing instance variables directly.

- We can write `print(cathal.name)` but this will use the `getName()` method to access the instance variable.
- We get the simplicity of the syntax but with error checking from the getter/setter.
- We can use two techniques: properties or property decorators.

We can use either method, but it's better to choose one and stick with it, rather than alternate.

Property Decorators Example

```
class Person:


    @property
    def pay(self):


        return self._pay


    @pay.setter
    def pay(self, pay):


        if pay < 0 or pay > 100000:
            print("%i is an invalid pay value - no value
set" %(pay))
        else:
            self._pay = pay


    if __name__ == '__main__':
        cathal = Person('Cathal', 'dev', 70000)
        cathal.pay = 76000
        print(cathal.pay)
```

The getter and setter functions are now just named "pay".

Property Decorators are nice because the property info is right next to the methods.

Properties Example

```
class Person:


    def getPay(self):
        return self._pay


    def setPay(self, pay):


        if pay < 0 or pay > 100000:
            print("%i is an invalid pay value - no value 
set" %(pay))
        else:
            self._pay = pay


    pay = property(getPay, setPay)  #Note the order of the 
methods in the list.
```

This is used the exact same way as Property Decorators.

Typical Class Definition Question

Question

Write a class - Square - to represent a square shape. The square should have the following state:

- length_of_side - the length of side of the square
- perimeter - the length of the perimeter of the square
- area - the area of the square

The class should provide a constructor to initialise instances. This should take one parameter - length_of_side - and should derive all other values from this. The length_of_side can be reset. Each of the three values can be read but only length_of_side can be set.

Once a base implementation has been written, convert it to use both types of property implementations.

Code

```
class Square(object):


    def __init__(self, length_of_side):


        # Could add error-handling here (We'll be seeing
this later)


        self._length_of_side = length_of_side
        self._perimiter = 4 * length_of_side
        self._area = length_of_side ** 2


    def getLengthOfSide(self):


        return self._length_of_side


    def setLengthOfSide(self, length_of_side):


        self._length_of_side = length_of_side
        self._perimiter = 4 * length_of_side
        self._area = length_of_side ** 2


    def getArea(self):


        return self._area


    def getPerimeter(self):


        return self._perimeter
    def __str__(self):

        descriptive_str = ("Side length: %i - Area: %i -
Perimeter: %i"
                            % (self._length_of_side,
self._area, self._perimeter))

def main():

    square1 = Square(4)
    print(square1)

    square1.setLengthOfSide(5)
    print(square1)

    print("Length: %i" % (square1.getLengthOfSide()))
    print("Area: %i" % (square1.getArea()))
    print("Perimeter: %i" % (square1.getPerimeter()))

if __name__ == '__main__':
    main()
```

Inheritance

How Do We Use Classes?

Take the development of a GUI. Classes that implement buttons and text boxes and all other components in a GUI can be reused time and time again.

We don't write all of the code that we need for a button each time.

We specialise the button by setting its state or perhaps adding behaviour, but the bulk of the code is reused time and time again.

This gives us some advantages:

- Parallel development in a team is now possible
- The code is simplified (or "less tightly bound")
- Easier bug fixing – if a bug exists, we fix it in the reponsible class and the fix propagates
- We can replace our code with better versions – if we develop a better class, as long as the class name and method headers are the same, we can replace the class and not break any code that uses it.
    - If we implemented the button from scratch every time, we would have had to replace and test the code in numerous places.

Mechanisms for Reuse

- Inheritance – allows us to take an existing class and specialise and extend it.
- Composition – allows us to divide a class into sub-classes, where the sub-class might be reused in a different implementation later.
    - E.g. a car has an engine, so a car object might contain an engine object.

Implementing Inheritance

Suppose we need to implement a payroll system for a company to manage its employees.

- Everyone has a name, social security number, and salary.
- Everyone can receive a pay rise, but this is calculated differently for managers.
- Engineers have a special skill, but managers don't.
- Managers are in charge of a project, while engineers are not.

We could write two classes, one for engineers and one for managers. This would be manageable for when there are two employee types, but we might needs to add other types, e.g. secretaries.

We use inheritance:

- Write an employee class that holds the common requirements for both engineers and managers.
- We then write two other classes for managers and engineers, which inherit from the employee class.

Some Terminology

- *Base Class / Super-class* – A more generic implementation containing state and behaviour common to sub-classes.
- *Derived Class / Sub-class* – A more specific implementation containing state and behaviour associated only with the derived class.
- *Single Inheritance* – Where a sub-class inherits from just a single super-class.
- *Multiple Inheritance* – Where a sub-class inherits from several super-classes.
- *Multi-level Inheritance* – Where a base class itself inherits from a more general base class.
- *Extension* – Addition of attributes or methods that implement functionality specific to the sub-class.
- *Specialisation* – Addition of a special version of a method already implemented in the base class.

The Search Heirarchy

When you inherit, you essentially provide a search path for methods and variables.

1. When the code runs, the sub-class is searched.
2. If the variable or method is not found there, the class it inherits from is searched.

3. This search process continues until the method or variable is found, or the search heirarchy is exhausted (there are no more super-classes to search).

This means that the most specific version of a method or variable is used. We can use this to override methods that already exist in super-classes (specialisation).

UML (Unified Modelling Language)

This is a design technique that provides a multi-faceted visualisation of software systems.

The class diagram provides a visualisation of a structure that describes the class heirarchy implemented by the system. It refers to OO concepts, which can be supported to various degrees by different languages.

UML Class Diagrams

UML class diagrams capture:

- Member variables and methods, and their scope and visibility.
- Relationships between classes and instances of those classes.

Key Steps for Inheritance

Inheriting

1. Ensure we can load the base class.
   `from employee import Employee`
2. Specify the class we wish to inherit from.
   `class Engineer(Employee):`

Initialising

- Call the constructor for the base class to ensure it is correctly initialised.

The derived class must set values for its instance variables while delegating to the constructor for the base class to ensure it is properly initialised.

```
def __init__(self, name, ssn, salary, skill):
    Employee.__init__(self, name, ssn, salary)
    self._skill = skill
```

Composition

When we create a new class, we're creating a new data type. Since we can make classes with sets of ints, floats, strings, etc., why not build classes unsing instances of objects we've created? This gives us the following advantages:

- We can create simpler, more maintable code.
- We can break out code that we use frequently into classes of functionality so that it can be reused.

For example, an iPhone 6 is a phone – it's got a screen, memory, a CPU. A laptop also has a screen, memory, and a CPU.

So it makes sense to write a Screen class, which could be used in both your Phone class and your Laptop class.

With Composition we get many of the same benefits as with Inheritance, but we're capturing a different relationship between objects.

In-Class Example

This will primarliy be an example of Inheritance, but will also have some Composition in it.

Requirements

- Model the data in a rental agreement.
- A rental agreement is made up of a contract and a property being rented. A contract consists of a rent amount and an expiry date (we'll treat the date as a string). A property can be a house or an apartment.
- Houses are described by their floor area, number of floors, address, and

whether or not they have a garage.
- Apartments are described by their floor area, address, apartment number, and whether or not they have a balcony.

UML (ish)

(This was drawn on the board, so this is only a rough translation.)

Within this requirements description, there are five things we'll want to model with objects:

- Property
    - floorarea: int
    - address: string
- House (Property)
    - nooffloors: int
    - hasGarage: bool
- Apartment (Property)
    - hasBalcony: bool
    - apartmentno: string
- Rental
    - prop: Property
    - contract: Contract
- Contract
    - expiry: string
    - rent: float

`House` and `Apartment` will both inherit from `Property`. This is indicated in a UML diagram with an arrow pointing from most specific to most generic.

Note the `Rental` object contains two objects – an instance of our `Property` class and another of our `Contract` class. This is indicated in a UML diagram with an arrow with a diamond head, either filled in or left empty (not sure which).

It would be common to come back and revise the initial design document during implementation, as you realise flaws of your approach and better ways to do things.

Refining

We need to override the getter for the address the `Apartment` class inherits from the `Property` class so that it can deal with combining the apartment number and the address.

Lab Assignment Left as an Exercise

There's an implementation of these classes on Moodle. Extend the design by adding another kind of property: office space.

Loose Ends

Polymorphism

When we write a class, we are creating a new type. While Python is dynamically typed, every object still has a type.

A good example of polymorphism is the multiplication operator in Python. This operates differently with two numbers as input and with one number and one string as input. This is polymorphism.

Polymorphism captures the concept that the effect of calling some object's method depends on the type of the object.

We also saw this when `givePayRaise()` operated differently depending on whether the object we called it with was a `Manager` or `Engineer` object, as we had written a specialised version of the method for our `Manager` class.

Duck Typing

Duck Typing is a name given to the process of establishing whether or not some object supports a particular behaviour or state.

I.e. If it walks like a duck and quacks like a duck, is it a duck?

[note to self: look it up]

Operator Overloading

Python allows us to define the meaning of operations built into Python when applied to our classes (e.g. `+`, `*`, `-`, `/`).

Often it doesn't make conceptual sense to overload these operators (e.g. what does it mean to subtract one student from another?), in which cases it shouldn't be done.

Sometimes it does make sense, though, e.g. checking if two `Student` objects refer to the same student with the `==` operator. It's often open to interpretation though. Generally in industry there are explicit methods rather than operator overloads for these cases.

To overload Python's operators, you use the specific method header. E.g. `__add__(self, other_student)` is called when we add two student objects. We must provide an implementation for this method if we want to be able to add instances of our objects to each other.

We've seen `__str__` and `__init__` already. They're called magic methods, and are identified by the double underscore.

Example Method Headers

- `__add__` - addition
- `__lt__` - less than
- `__eq__` - equivalent
- `__ne__` - not equivalent

What does equivalence mean? It depends on the context. Often methods such as `hasSameID` are used instead because they're more descriptive.

With `==`, the equivalence method of the object on the left is called, passing the object on the right as a parameter.

Object Persistence

There are a number of ways to persist data from your program:

- Write to files
- Store data in a database
- Serialise your objects and write them to disk (shelving)

Pickles and Shelves

A pickle allows us to serialise any object or collection of objects (converts them to a string of bytes and writes them to a file).

Shelves allow us to store pickled objects as key/value pairs. They function like dictionaries but we must open and close them (read to and write from the disk).

Pickling is done under the hood when we shelve an object – we don't have to explicitly do this.

Using Shelve

```
import shelve
from employee import Employee

john = Employee("SN12345", "John Doe", 32000)



db = shelve.open("employeedb")


db[john.ssn] = john


db.close()
```

We can also store e.g. a list of objects in the same way.