Introduction to OOP – Part 1

COMP16321 – Programming 1

Terence Morley

Department of Computer Science

The University of Manchester

Objectives for this Lecture

To understand and be able to use the following concepts in your Python code:

- Classes and Objects
- Instance Attributes and Methods
- Encapsulation
- Class Attributes and Methods
- Inheritance
- Object-oriented Design

Imperative Programming

Imperative Programming is what you have been doing so far:

```
x = 2
y = 4
z = x + y

print(x, '+', y, '=', z)
```

That is, a set of commands

In grammar, the *imperative* is the form of a verb for giving an order or command

Imperative Programming

If we did our programming with a big list of commands,
it would be difficult to read and to manage

So, we normally create functions (or subroutines or procedures) to make our code more modular:

```
def sum(x, y):
        return x + y
```

This is known as **Procedural Programming**

and is a type of imperative programming

Object-Oriented Programming

Object-oriented Programming (OOP) and Object-oriented Design (OOD)

is an programming paradigm based on **objects**

(paradigm = a set of concepts)

Objects are distinct entities that have attributes and can perform actions

Examples of objects are Student, Bank Account, Engine Management System

Classes and Objects

Terminology

We have just described things like Student as an object

In Object-oriented Programming (OOP) terminology,

these are actually **classes** of object

The **objects** are the **instances** of these classes

So, John Smith is an object, which is an instance of the class Student

Attributes

Class	Example Object	Example Attributes
Student	susan_stevens	name age address
BankAccount	current_bank_account	account_number owner balance overdraft_limit
EngineManagementSystem	ford_focus_ems	engine_temperature engine_speed fuel_flowrate

Attributes

You will also come across **attributes** being called

properties or **member variables**

These terms can have subtle differences but people will use them interchangeably

(See later for information on properties)

Imperative Programming – Problems

You might think that you are already doing Object-oriented Programming

You have identified classes, e.g. Student

and you have identified the attributes: name, age, address, etc.

so you can store the data like this:

```
student_names = [...]
student_ages = [...]
student_addresses = [..]
```

Well, you are thinking in an Object-oriented way, but that is not OOP

Imperative Programming – Problems

In imperative programming we might implement attributes of a student in different variables:

```python
student_names = ["Larken Rose",
                "Julie Smith",
                "Brian Ferry"]
student_ages = [20, 21, 19]
student_addresses = ["2023 Jones Plantation",
                "14 Cliff Lane",
                "192 Long Road"]

def print_student_info(index):
    print(student_names[index])
    print(student_ages[index])
    print(student_addresses[index])
```

But what about manipulations such as sorting and appending/inserting/deleting?

Imperative Programming – Problems

We could use a list of lists (lists can contain different data types)

```
students = [
            ["Larken Rose", 20, "2023 Jones Plantation"],
            ["Julie Smith", 21, "14 Cliff Lane"],
            ["Brian Ferry", 19, "192 Long Road"]
            ]

def print_student_info(index):
    print(students[index][0])
    print(students[index][1])
    print(students[index][2])
```

But now we need to access attributes by index rather than name

OOP

We will now see that Python gives us ways to actually program in a Object-oriented way

MANCHESTER
1824
The University of Manchester

Introduction to OOP – Part 2

COMP16321 – Programming 1

Terence Morley

Department of Computer Science

The University of Manchester

OOP – Attributes

In OOP, we declare the attributes within the class

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

Python executes the `__init__()` function when an object is instantiated

We can instantiate an object (create it in memory) as follows:

```
student = Student("Larken Rose", 20, "2023 Jones Plantation")
```

OOP – Attributes

Notice that the declaration of `__init__()` contains the variable `self`

```
class Student:
    def __init__(self, name, age, address):
        pass
```

But nothing is passed in to that variable by the programmer when instantiating an object:

```
student = Student("Larken Rose", 20, "2023 Jones Plantation")
```

OOP – Attributes

Creating a list of Student objects and accessing the attributes:

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

students = [
    Student("Larken Rose", 20, "2023 Jones Plantation"),
    Student("Julie Smith", 21, "14 Cliff Lane"),
    Student("Brian Ferry", 19, "192 Long Road")
    ]

def print_student_info(index):
    print(students[index].name)
    print(students[index].age)
    print(students[index].address)
```

OOP – Case

Notice the case that was used for the identifiers:

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


student = Student("Larken Rose", 20, "2023 Jones Plantation")
```

Class names are usually written in Pascal Case (or CapWords):

`Student, BankAccount`

Variable names and **function names** are normally written in Snake Case (lowercase with words separated by underscores):

`age, first_name, delete_user()`

OOP – Methods

We have just seen the `__init__()` function within the class

This shows us that we can also put functions inside classes

Therefore, a class is not just a structure (like in the C language) that allows us to keep a set of related variables together

It allows us to implement the functionality of the class inside its definition

In other languages, this initialisation function would be called a **constructor**

OOP – Methods

Adding a print() function to the class

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

    def print(self):
        print(self.name)
        print(self.age)
        print(self.address)
```

```
student1 = Student("Kate Bush", 32, "16 Wuthering Heights")

# Print by accessing the attributes
print(student1.name, student1.age, student1.address)

# Print via the class method
student1.print()
```

OOP – Methods

Alternative way of accessing the print() function

```
student1 = Student("Kate Bush", 32, "16 Wuthering Heights")

# Print by accessing the object through the class
Student.print(student1)
```

So, in this case we are passing something into the `self` parameter

__str__ Method

If we create an object and try to print it as shown below:

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


if __name__ == '__main__':
    student1 = Student("Kate Bush", 32, "16 Wuthering Heights")

    print(student1)
```

Python just prints out the type and address of the object:

```
<__main__.Student object at 0x7f8f25082b00>
```

__str__ Method

However, if we add a `__str__` method to the class:

```
class Student:
    ...

    def __str__(self):
        return self.name + '\n' + str(self.age) + '\n' + self.address


if __name__ == '__main__':
    student1 = Student("Kate Bush", 32, "16 Wuthering Heights")

    print(student1)
```

Python calls the `__str__` function in our class and prints out what we tell it to:

```
Kate Bush
32
16 Wuthering Heights
```

UML Class Diagram

Unified Modelling Language (UML) is a popular diagramming method for the design of software

The class we have created so far will have a diagram that looks like this:

Student
+name: string +age: int +address: string
+__init__(name: string, age: int, address: string): Student +print(): void

The '+' means that the attribute or method is **public** (see later)

Introduction to OOP – Part 3

COMP16321 – Programming 1

Terence Morley

Department of Computer Science

The University of Manchester

Encapsulation

Encapsulation means the combining of data (attributes) and functionality (methods) into objects

and limiting the direct access to its internals

We have already covered the first part of this definition

Data-hiding will be described next

Public and Private Access

We can access attributes directly from our classes, as we have seen:

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


if __name__ == '__main__':
    student1 = Student("Terry Morley", 21, "1 Small Avenue")
    print(student1.name)

    student1.name = "John Jones"
    print(student1.name)
```

```
Terry Morley
John Jones
```

The variable `name` is **public** (remember the '+' in the UML diagram)

Public and Private Access

If we precede our variable name with two underscores (`self.__name`) it becomes a **private** variable (indicated with a '−' in the UML diagram):

```
class Student:
    def __init__(self, name, age, address):
        self.__name = name


if __name__ == '__main__':
    student1 = Student("Terry Morley", 21, "1 Small Avenue")
    print(student1.__name)
```

Output:

```
  File "public_private2.py", line 10, in <module>
    print(student1.__name)
AttributeError: 'Student' object has no attribute '__name'
```

Getters

We then need to use methods to access the variable

```
class Student:
    def __init__(self, name, age, address):
        self.__name = name

    def get_name(self):
        return self.__name

if __name__ == '__main__':
    student1 = Student("Terry Morley", 21, "1 Small Avenue")
    print(student1.get_name())
```

The method `get_name()` is called a **getter**

This lets us to do some processing before providing the caller with the data — we might want to combine a forename and surname, for example

Setters

Similarly, we need a method (a **setter**) to allow us to set the value of a private attribute:

```
class Student:
    def __init__(self, name, age, address):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name


if __name__ == '__main__':
    student1 = Student("Terry Morley", 21, "1 Small Avenue")
    student1.set_name("Jimmy White")
    print(student1.get_name())
```

UML Class Diagram

Our UML class diagram will now look like this:

Student
-name: string -age: int -address: string
+__init__(name: string, age: int, address: string): Student +get_name( ): string +set_name(name: string): void +get_age( ): int +set_age(age: int): void +get_address( ): string +set_address(address: string): void

Setters

Why would we want a setter?

It allows us to do some processing before storing the data in the object

For example:

- Validate the data (e.g. if the attribute is a postcode, check that it's valid)
- Perform other actions (for example, also store the attribute in a database)

Properties

Maybe you think it's a bit cumbersome to use get_name() and set_name()

Fortunately, Python lets you implement getters and setters in a more friendly way while still giving you control over what happens

We can access our private `__name` attribute through a **property** called `name`

(So, a property isn't exactly the same as an attribute)

@property and @setter

The getter and setter from an earlier slide are modified slightly as well as adding the @property and @name.setter **decorators**

```
class Student:
    def __init__(self, name, age, address):
        self.__name = name

    @property
    def name(self):
        return self.__name

    @name.setter
    def name(self, name):
        self.__name = name

if __name__ == '__main__':
    student1 = Student("Terry Morley", 21, "1 Small Avenue")
    student1.name = "Jimmy White"
    print(student1.name)
```

Private Methods

In the same way as we create private attributes (prefixing the name with __), we can also create private methods

You might want to do this, for example, on a method that writes a record to a database

You might only want your class to write the record under your control
You might not want a user of your class doing it

Private Methods

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

    def change_details(self, name, age):
        self.name = name
        self.age = age
        self.__update_db()

    def __update_db(self):
        print('TODO: update the database with', self.name, self.age)


if __name__ == '__main__':
        person = Person('Penny Lane', 67)
        person.change_details('Penny Smith', 67)

        print('\nAttempting to access private method outside an object:\n')
        person.__update_db()
```

Private Methods

Output:

```
TODO: update the database with Penny Smith 67

Attempting to access private method outside an object:

Traceback (most recent call last):
  File "/comp16321_oop_code/private_method.py", line 23, in <module>
    person.__update_db()
    ^^^^^^^^^^^^^^^^^^
AttributeError: 'Person' object has no attribute '__update_db'
```

Encapsulation

Imagine that we have written our class code into libraries that we make available to developers

Now that we have made attributes private and let users access them through properties,

and we have also made some functions private,

we have gone some way towards **hiding the implementation details** from the users of our code

Encapsulation

Why do we need to hide the implementation details?

As an example, imagine that we haven't used private attributes

We made a Student class and we expect developers to set the name and age through the initialiser

But a developer decides that he/she wants to change the age after creating the object

They look in your code and see the age attribute and decide to set it themselves...

Encapsulation

Now you decide that storing the age isn't a good idea. It is better to store the year of birth

So, you leave the initialiser accepting a name and age, but you calculate the year of birth in the initialiser and store that instead

You then release a new version of your library

Unfortunately, this will break the code of the developer who directly accessed your age attribute

If you hide the implementation details, you can change the internals of your classes without the risk of breaking anyone else's code

The University of Manchester

Introduction to OOP – Part 4

COMP16321 – Programming 1

Terence Morley

Department of Computer Science

The University of Manchester

Other Attribute and Method Types

We have seen attributes and methods that apply to objects

Attributes such as the name and age of a particular Person object
Methods such as for printing the attributes of an object

Next, we will consider attributes and methods that apply to the classes

Class Attributes

Say we want keep track of the number of Student objects
that have been instantiated

We could just keep a count somewhere in our program,
but it would be neater to store it in the class somehow

Class Attributes

```
class Student:
    number_of_students = 0

    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address
        Student.number_of_students += 1


if __name__ == '__main__':
    student1 = Student("Larken Rose", 20, "2023 Jones Plantation")
    student2 = Student("Julie Smith", 21, "14 Cliff Lane")
    student3 = Student("Brian Ferry", 19, "192 Long Road")
    print('Total number of students:', Student.number_of_students)
```

Output:

```
Total number of students: 3
```

Class Attributes

Note that we can access the class attribute through the class or an object:

```
print('Total number of students:', Student.number_of_students)
print('Total number of students:', student1.number_of_students)
```

Class Methods

We might also want to make the class attribute private by putting two underscores in front of the name: __number_of_students

We then need a method that can access this private class attribute

For this we use a class method that requires the decorator @classmethod

Class Methods

```
class Student:
    __number_of_students = 0

    def __init__(self, name, age, address):
        Student.__number_of_students += 1

    @classmethod
    def get_number_of_students(cls):
        return Student.__number_of_students

if __name__ == '__main__':
    student1 = Student("Larken Rose", 20, "2023 Jones Plantation")
    student2 = Student("Julie Smith", 21, "14 Cliff Lane")
    student3 = Student("Brian Ferry", 19, "192 Long Road")
    print('Total number of students:', Student.get_number_of_students())
```

Class Methods

Another use for a class method is as a **factory method**

A factory method is on that can create new instances

We have created a Student class that is initialised with name, age, and adddress

We might also want to allow the name to be specified as a first name and family name

A factory method lets us do this

Class Methods

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

    @classmethod
    def from_separate_names(cls, first_name, family_name, age, address):
        return cls(first_name + " " + family_name, age, address)

if __name__ == '__main__':
    student1 = Student("Larken Rose", 20, "2023 Jones Plantation")
    student2 = Student.from_separate_names("Julie", "Smith", 21, "14 Cliff Lane")

    print(student1.name)
    print(student2.name)
```

Output:

```
Larken Rose
Julie Smith
```

Static Methods

Sometimes you might want a function that is, in some way, related to a class

but doesn't need an object to be instantiated to use it

and doesn't need to know anything about the class

Static Methods

As an example, say the Student class contains a student ID which also shows their department

(MA25542 = a mathematics student, CS77421 = a computer science student)

The Student class could have a function that accepts an ID and checks if they are a computer science student

A static method could be used for this

Static Methods

```
class Student:
    def __init__(self, name, age, address):
        pass

    @staticmethod
    def is_cs_student(id):
        return id.startswith('CS')

if __name__ == '__main__':
    print(Student.is_cs_student('MA25542'))
    print(Student.is_cs_student('CS77421'))}
```

Output:

```
False
True
```

MANCHESTER 1824

The University of Manchester

Introduction to OOP – Part 5

COMP16321 – Programming 1

Terence Morley

Department of Computer Science

The University of Manchester

Inheritance

Pets

Dog	Cat	Mouse	Snake
avg_price	avg_price	avg_price	avg_price
hair_colour	hair_colour	hair_colour	skin_colour
num_walks	has_tail	min_cage_size	poisonous
bark()	meow()	squeak()	hiss()
run()	spring()	scurry()	slither()
eat()	eat()	eat()	eat()

Inheritance

Notice that all of the classes have a common attribute of avg_price

and they have the common method, eat()

Consider that the eat() method might have the exact same functionality for each tpe of pet

If we decide to modify the functionality of eat() we would need to do it in every class

Inheritance

I would be good if we could put these attributes and methods into a separate class and let the other classes use them:

Pet
avg_price
eat()

You might now think ‘Oh, yes, I can just include a Pet object in my Mouse class’

You could do that. That is called **Aggregation** (see next slide)

But we will soon see that **Inheritance** is more powerful
(and will make more sense in this case, see later)

Aggregation

```
class Pet:
    def __init__(self, avg_price):
        self.avg_price = avg_price

    def eat(self):
        print( "Munch, munch.")

class Mouse:
    def __init__(self, avg_price):
        self.pet = Pet(avg_price)

    def scurry(self):
        print("I'm scurrying.")


if __name__ == '__main__':
    mouse = Mouse(4.5)
    mouse.scurry()
    mouse.pet.eat()
```

Inheritance

While we're thinking about simlifying things:

Each of our pet classes has a method that makes some kind of sound:
bark(), meow(), etc.

We could add a `sound` attribute to the Pet class (sound = 'Woof')
and also add a method called `make_sound()`

Inheritance

Our new Pet class

Pet
avg_price sound
make_sound() eat()

Inheritance

Earlier, we used **Aggregation** by putting a Pet object inside a Mouse object

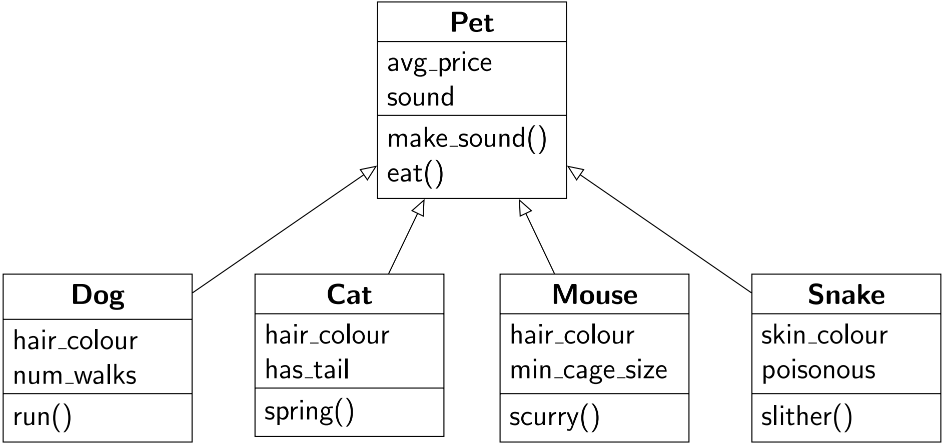
That is saying that a mouse **contains** a pet

But that's not right

What we are really thinking, is that a mouse **is** a pet

Python **Inheritance** allows us to make this '**is-a**' relationship

Inheritance



Different types of pet **derived** from the Pet class

Different types of pet **inherit** the functionality from the Pet class

Inheritance

```python
class Pet:
    def __init__(self, avg_price):
        self.avg_price = avg_price

    def eat(self):
        print( "Munch, munch.")

class Mouse(Pet):
    def __init__(self, avg_price):
        Pet.__init__(self, avg_price) # Call initialiser of the Base Class

    def scurry(self):
        print("I'm scurrying.")


if __name__ == '__main__':
    mouse = Mouse(4.5)
    mouse.scurry()
    mouse.eat()
    print("{} UK Pounds".format(mouse.avg_price))
```

Overriding

We put the eat() method in the Pet base class because we expected it to be the same for all types of pet (Munch, munch)

But what if we decide that Dogs are a bit different to all other pets

In this case, we can declare an eat() method in the Dog class and this **overrides** the eat() method in the Pet class

This means that when the eat() method is called on a Dog object, it calls the eat() method in the derived Dog class instead of the one in the base class

Overriding

```python
class Pet:
    def __init__(self, avg_price):
        self.avg_price = avg_price

    def eat(self):
        print( "Munch, munch.")

class Dog(Pet):
    def __init__(self, avg_price):
        Pet.__init__(self, avg_price)

    def eat(self): # Overrides the function in the base class
        print("Gulp, munch, slurp, splash, snort.")


if __name__ == '__main__':
    fido = Dog(4.5)
    fido.eat()
```

OOP

You should now be able to make your code more structured
with all of the data and functionality for an object within the class

Object Oriented Design (OOD)

When designing an application or library,

how do we find out what classes we should create

and what attributes and methods they require?

OOD – Requirements Analysis

User Story

As an order taker, I answer phone calls from customers and take their orders. If they are an existing customer, I need to see their details: the company name, company address, delivery address if different, contact name, contact phone number and email address. If they are a new customer, I need to create a new account for them. Then I need to see if they have made any recent orders – maybe they have got a query about a delivery or want to make a payment. If so, I will take their debit card number, expiry date and security code.
If they want to make an order, I want to record that on the system. The customer will give me product codes and I want to be able to see the details of those products, such as the name, description, price, and if the product is in stock...

OOD – Requirements Analysis

We can then examine the user stories and highlight nouns and verbs

The nouns might show us classes or class attributes

The verbs might show us the class methods

OOD – Requirements Analysis

User Story

As an order taker, I answer phone calls from **customers** and **take** their **orders**. If they are an existing customer, I need to **view** their details: the **company name**, **company address**, **delivery address** if different, **contact name**, **contact phone number** and **email address**. If they are a new customer, I need to **create** a new **account** for them. Then I need to **see** if they have made any **recent orders** – maybe they have got a query about a **delivery** or want to **make** a **payment**. If so, I will take their **debit card number**, **expiry date** and **security code**. If they want to make an order, I want to **record** that on the system. The customer will give me **product codes** and I want to be able to **see** the details of those **products**, such as the **name**, **description**, **price**, and if the product is in stock...

OOD – Requirements Analysis

We can then start desiging classes:

Customer
company_name
address
contact_name
contact_phone
__init__(name, address): Customer
show_details()
show_recent_orders()

- The classes can be coded directly from the class diagram
- The methods can be empty (just contain `pass`) until more of the classes are developed

Summary

What have we discussed?

- Classes and Objects
- Instance Attributes and Methods
- Encapsulation
- Class Attributes and Methods
- Inheritance
- Object-oriented Design