Python

Introduction to Programming

Comp07027

Lecture 14

OOP 4

Inheritance & Encapsulation

What has OOP ever done for us??

Object Oriented Program has three very useful properties:

- **Polymorphism**
- **Inheritance**
- **Encapsulation**

Polymorphism

We have already seen (and used) polymorphism.

Polymorphism is ability to treat different objects in the same way e.g. different methods with the same name in different objects.

```
class square:
    def __init__(self, side):
        self.side = side
    def calculate_area(self):
        return self.side ** 2

class circle:
    def __init__(self, radius):
        self.radius = radius
    def calculate_area(self):
        return 3.14 * (self.radius ** 2)

list_of_shapes = [circle(3), square(5), circle(4), square(2)]

for each_shape in list_of_shapes:
    print("Area = ", each_shape.calculate_area())
print("All done")
```

```
"C:\Program Files\Pytho
Area =  28.26
Area =  25
Area =  50.24
Area =  4
All done
```

What has OOP ever done for us??

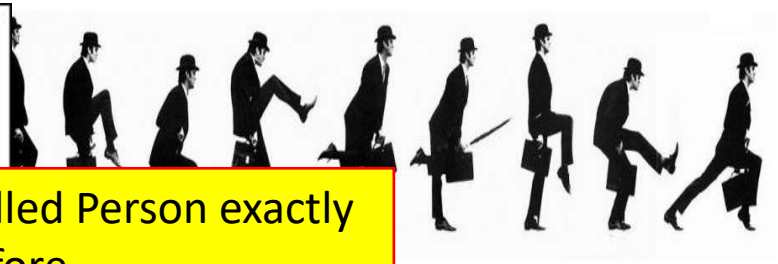
Let’s look at **Inheritance** now

Inheritance

Inheritance allows us to define a class that inherits all the methods and properties from another class.

```
class Person:
    def __init__ (self, first_name, surname):
        self.first_name = first_name
        self.surname = surname

    def display_name(self):
        print("You must be", self.first_name, self.surname )

class Student(Person):
    def __init__ (self, first_name, surname, university):
        super().__init__(first_name, surname)
        self.university = university

    def welcome(self):
        print("Well, welcome to", self.university, self.first_name)

student1 = Student("Eric", "Idle", "UWS")
student1.display_name()
student1.welcome()
```

```python
class Person:
    def __init__ (self, first_name, surname):
        self.first_name = first_name
        self.surname = surname

    def display_name(self):
        print("You must be", self.first_name, self.surname )

class Student(Person):
    def __init__ (self, first_name, surname, university):
        super().__init__(first_name, surname)
        self.university = university

    def welcome(self):
        print("Well, welcome to", self.university, self.first_name)

student1 = Student("Eric", "Idle", "UWS")
student1.display_name()
student1.welcome()
```



We create a class called Person exactly as we have done before.
Person has a first name and a surname, and a method to display them.

```python
class Person:
    def __init__ (self, first_name, surname):
        self.first_name = first_name
        self.surname = surname

    def display_name(self):
        print("You must be", self.first_name, self.surname )

class Student(Person):
    def __init__ (self, first_name, surname, university):
        super().__init__(first_name, surname)
        self.university = university

    def welcome(self):
        print("Well, welcome to", self.university, self.first_name)

student1 = Student("Eric", "Idle", "UWS")
student1.display_name()
student1.welcome()
```

We create a class called Person exactly as we have done before.
Person has a first name and a surname, and a method to display them.

We now add a class called Student (which is a type of person!).
Student has a first name, a surname and a university, and a method to display the first name and university.

```python
class Person:
    def __init__ (self, first_name, surname):
        self.first_name = first_name
        self.surname = surname

    def display_name(self):
        print("You must be", self.first_name, self.surname )

class Student(Person):
    def __init__ (self, first_name, surname, university):
        super().__init__(first_name, surname)
        self.university = university

    def welcome(self):
        print("Well, welcome to", self.university, self.first_name)

student1 = Student("Eric", "Idle", "UWS")
student1.display_name()
student1.welcome()
```

Notice that our declaration of Student is different:
class student(person)
This creates a relationship by passing Person to Student as a parameter. Person is the **PARENT** class and Student is the **CHILD** class. The child class **INHERITS** from the parent class i.e. Student can use the properties and methods from Person.

```python
class Person:
    def __init__ (self, first_name, surname):
        self.first_name = first_name
        self.surname = surname

    def display_name(self):
        print("You must be", self.first_name, self.surname )

class Student(Person):
    def __init__ (self, first_name, surname, university):
        super().__init__(first_name, surname)
        self.university = university

    def welcome(self):
        print("Well, welcome to", self.university, self.first_name)

student1 = Student("Eric", "Idle", "UWS")
student1.display_name()
student1.welcome()
```

However, Student can also add its own properties and methods: Student gets **first_name** and **surname** from Person, but **university** is from Student.

```python
class Person:
    def __init__ (self, first_name, surname):
        self.first_name = first_name
        self.surname = surname

    def display_name(self):
        print("You must be", self.first_name, self.surname )

class Student(Person):
    def __init__ (self, first_name, surname, university):
        super().__init__(first_name, surname)
        self.university = university

    def welcome(self):
        print("Well, welcome to", self.university, self.first_name)

student1 = Student("Eric", "Idle", "UWS")
student1.display_name()
student1.welcome()
```

We can now create a Student object, *student1*, with a first name, surname and university. Then we can call the methods

display_name from **Person**

welcome from **Student**

```python
class Person:
    def __init__ (self, first_name, surname):
        self.first_name = first_name
        self.surname = surname

    def display_name(self):
        print("You must be", self.first_name, self.surname )

class Student(Person):
    def __init__ (self, first_name, surname, university):
        super().__init__(first_name, surname)
        self.university = university

    def welcome(self):
        print("Well, welcome to", self.university, self.first_name)

student1 = Student("Eric", "Idle", "UWS")
student1.display_name()
student1.welcome()
```

```
You must be Eric Idle

Well, welcome to UWS Eric
```

We can now create a Student object, *student1*, with a first name, surname and university. Then we can call the methods

display_name from **Person**

welcome from **Student**

What has OOP ever done for us??

Finally we look at **Encapsulation**

Encapsulation

We create a class called **user**.

The class has two attributes – **username** and **password**.

We define methods to **set** (change) the password, and another two to **get** the password, and **get** the username.

```
class user:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def set_password(self):
        self.password = input("Enter NEW password >  ")

    def get_password(self):
        return self.password

    def get_username(self):
        return self.username


user1 = user("MichaelPalin", "P4rr0t")


print()
print("Username and Password")
print("--------------------")
print("username is:  ", user1.get_username())
print("password is:  ", user1.get_password() )
print()
```

Encapsulation

We also create a method to display the details!

We do not usually do this, it is merely included to let us see what is happening while we are coding.

If we run the program ....

```
class user:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def set_password(self):
        self.password = input("Enter NEW password >  ")

    def get_password(self):
        return self.password

    def get_username(self):
        return self.username


user1 = user("MichaelPalin", "P4rr0t")

print()
print("Username and Password")
print("--------------------")
print("username is:  ", user1.get_username())
print("password is:  ", user1.get_password() )
print()
```

Encapsulation

We also create a method to display the details!

We do not usually do this, it is merely included to let us see what is happening while we are coding.

If we run the program ....

```
class user:
    def __init__(self, username, password):
```

```
Username and Password
---------------------
username is:   MichaelPalin
password is:   P4rr0t
```

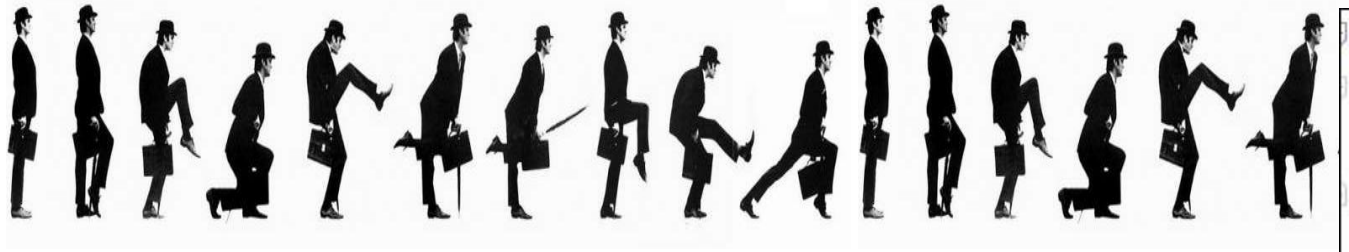
```
user1 = user("MichaelPalin", "P4rr0t")

print()
print("Username and Password")
print("---------------------")
print("username is:  ", user1.get_username())
print("password is:  ", user1.get_password() )
print()
```

Encapsulation

This is clearly not very secure!!!!

Any programmer can display a user's username and password.

```
Username and Password
---------------------
username is:   MichaelPalin
password is:   P4rr0t
```

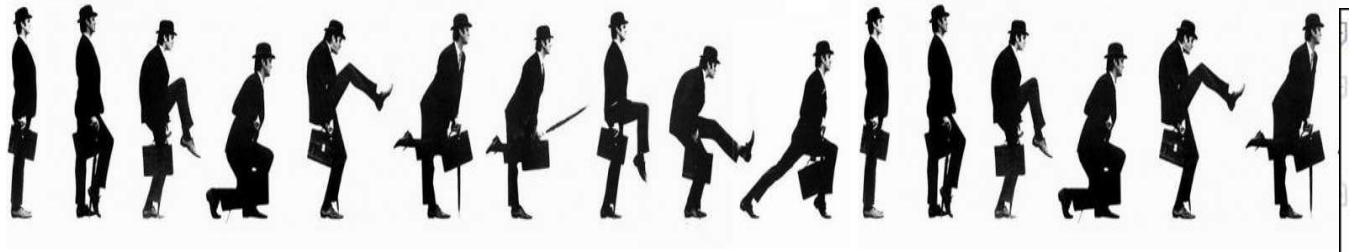


Encapsulation

If we make a slight change (**put two underscores before the methods' names**) we can make them private.

Now …..

```
class user:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def set_password(self):
        self.password = input("Enter NEW password >  ")

    def __get_password(self):
        return self.password

    def __get_username(self):
        return self.username


user1 = user("MichaelPalin", "P4rr0t")

print()
print("Username and Password")
print("--------------------")
print("username is:  ", user1.__get_username())
print("password is:  ", user1.__get_password() )
print()
```



Encapsulation

If we make a slight change (**put two underscores before the methods' names**) we can make them private.

Now …..

__get_username() is not recognised because it's hidden

```
class user:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def set_password(self):
        self.password = input("Enter NEW password >  ")

    def __get_password(self):
        return self.password

    def __get_username(self):
```

```
Traceback (most recent call last):

  File "C:/Users/22222250/Desktop/MemStick/OOP/password.py", line 21, in <module>
Username and Password
    print("username is:  ", user1.__get_username())
--------------------
AttributeError: 'user' object has no attribute '__get_username'
```

```
print("---------------------")
print("username is:  ", user1.__get_username())
print("password is:  ", user1.__get_password() )
print()
```

Encapsulation

We have now added two methods, one to change the password, one to display (to help with coding).

These methods can access the hidden methods because they are also inside the class.

Now when we run the program ….

```
class user:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def set_password(self):
        self.password = input("Enter NEW password >  ")

    def __get_password(self):
        return self.password

    def __get_username(self):
        return self.username

    def change_password(self):
        my_password = input("Enter your CURRENT password >  ")
        if my_password == user.__get_password(self):
            self.set_password()

    def display_details(self):
        print()
        print("Username and Password")
        print("--------------------")
        print("username is:  ", user.__get_username(self))
        print("password is:  ", user.__get_password(self))
        print()

user1 = user("MichaelPalin", "P4rr0t")

user1.display_details()
user1.change_password()
user1.display_details()
```

Encapsulation

We have now a
one to change t
to display (to h

These methods
hidden method
also inside the

Now when we run the program ....

```
Username and Password
--------------------
username is:   MichaelPalin
password is:   P4rr0t

Enter your CURRENT password >  P4rr0t
Enter NEW password >  fdse

Username and Password
--------------------
username is:   MichaelPalin
password is:   fdse
```

```python
class user:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def set_password(self):
        self.password = input("Enter NEW password >  ")

    def __get_password(self):
        return self.password

    def __get_username(self):
        return self.username

    def change_password(self):
        my_password = input("Enter your CURRENT password >  ")
        if my_password == user.__get_password(self):
            self.set_password()

    def display_details(self):
        print()
        print("Username and Password")
        print("--------------------")
        print("username is:  ", user.__get_username(self))
        print("password is:  ", user.__get_password(self))
        print()

user1 = user("MichaelPalin", "P4rr0t")

user1.display_details()
user1.change_password()
user1.display_details()
```

Encapsulation

So, what have we achieved??

We can now create code which will access the password from outside the object and allow us to check it without revealing it!

That’s much more secure.

Let’s make the program a bit more flexible …..

Encapsulation

Instead of individual variables (user1, user2) let's put all the instances of user in a list called **users[]**

```
class user:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def set_password(self):
        self.password = input("Enter NEW password >  ")

    def __get_password(self):
        return self.password

    def __get_username(self):
        return self.username

    def change_password(self):
        my_password = input("Enter your CURRENT password >  ")
        if my_password == user.__get_password(self):
            self.set_password()

    def display_details(self):
        print()
        print("Username and Password")
        print("--------------------")
        print("username is:  ", user.__get_username(self))
        print("password is:  ", user.__get_password(self))
        print()

users = [user("MichaelPalin", "P4rr0t"), user("EricIdle","M0nty"), user("TerryJones","Pyth0n")]
print(type(users))
users[0].display_details()
users[0].change_password()
users[0].display_details()
```

Encapsulation

Instead of individual variables (user1, user2) let's put all the instances of user in a list called **users[]**

Now we can append, find, add, delete, pickle and lots of other fun stuff!!!

```
class user:
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def set_password(self):
        self.password = input("Enter NEW pas

    def __get_password(self):
        return self.password

    def __get_username(self):
        return self.username

    def change_password(self):
        my_password = input("Enter your CURR
        if my_password == user.__get_passwor
            self.set_password()

    def display_details(self):
        print()
        print("Username and Password")
        print("--------------------")
        print("username is:  ", user.__get_username(self))
        print("password is:  ", user.__get_password(self))
        print()

users = [user("MichaelPalin", "P4rr0t"), user("EricIdle","M0nty"), user("TerryJones","Pyth0n")]
print(type(users))
users[0].display_details()
users[0].change_password()
users[0].display_details()
```

```
<class 'list'>

Username and Password
--------------------
username is:   MichaelPalin
password is:   P4rr0t

Enter your CURRENT password >  P4rr0t
Enter NEW password >  De4d

Username and Password
--------------------
username is:   MichaelPalin
password is:   De4d
```

For next week

Exercise 1

Write a program to demonstrate what **inheritance** is.

Use Animals, Felines and Lions.

This will require three layers!!

For next week

Exercise 2

Create the username/password example from the slides.

Add a menu at the front which allows us to

- Add a new user
- Delete a user
- Change the password

For next week

Exercise 3

Create the username/password example from the slides.

Add a menu at the front which allows us to

- Add a new user
- Delete a user
- Change the password

Now PICKLE it

Questions??