Rachel Evans

THEORY QUESTIONS ASSIGNMENT 2 - Software Stream

How does Object Oriented Programming differ from Process Orientated Programming?

There are a number of different approaches to the process of writing computer programs. These approaches are known as programming paradigms. Each of these approaches is a fundamentally different way of creating a solution to a specific type of problem. Two of the most important paradigms are the procedural programming paradigm and the object oriented paradigm.

Procedural programming, also known as imperative programming, uses a set of instructions to tell the computer what to do. As the name suggests it relies on procedures, (also known as routines, subroutines or functions) as a series of steps to be followed. During the execution of the program any given procedure might be called at any point. This approach is fairly intuitive, in that this is how you might expect a computer program to work. Most of the earliest computer programming languages, such as C, Fortran and COBOL, which have been around since the 1960s and 1970s are procedural. They are also known as top-down languages.

Object oriented programming or OOP is different from procedural programming in that objects rather than procedures are the basic units of the program. An object is a part of the program which contains data (known as attributes or properties) and code which tells it how to perform certain actions and interact with other objects in the program. This code is known as a method. Each object is an instance of a class, or a blueprint which contains all the details or fields of what each object will contain. A class is a concept and an object is the embodiment or instantiation of the concept. This paradigm strongly correlates with objects which exist in the real world. For example, an object could be a person. A person has a name - this would be considered an attribute of the person - and is able to do certain things, such as walk - this is considered a method of the person. A person class would provide a blueprint for everything a person would have and do.

The key differences between procedural programming and OOP are:

- OOP is underpinned by four key concepts - abstraction, inheritance, encapsulation and polymorphism. Procedural programming has no such concepts.
- Due to abstraction data hiding is possible in OOP which adds a level of security. As this doesn't exist in procedural programming it is much less secure than programs which use OOP.
- OOP is a bottom up approach, procedural programming is a top down approach.
- It can be difficult to find, maintain and modify functions, whereas in OOP it is easy to find and maintain existing code.
- In OOP access modifiers, private, public and protected exist. There are no such modifiers in procedural programming.

- In OOP new data objects can be easily created from existing data objects. There is no simple process to add data in procedural programming.

What is Polymorphism in OOP?

The word polymorphism comes from the ancient Greek - poly=many and morphism=forms and refers to when something exists in different forms. In computer programming polymorphism is one of the four core concepts of OOP (along with abstraction, encapsulation and inheritance). It refers to the ability of a function of the same name to carry out different functionality altogether, creating a structure which can use many forms of objects. That is, one method can have multiple implementations and which one is to be used is decided at runtime depending on the situation e.g. data type of the object. A good example is the len() function which can be used on several data types - lists, strings and dictionaries. The type of data it returns depends on the type of object it has been used on e.g. it returns the length of a string, the number of items in a list and the number of keys in a dictionary. Another example is the addition operator +. This can be used both to add integers together and to concatenate strings (e.g. 1 + 4 = 5, “one” + “four” = “onefour”.)

In OOP there are two types of polymorphism:

Static binding (or compile-time polymorphism) - e.g. method overloading
Dynamic binding (or runtime polymorphism) e.g. method overriding

In method overloading multiple methods can have the same name with different parameters, either a different number of parameters or changing the data type which can be passed as an argument.

E.g

```
class Animal:

   def sayHello(self, name=None):

        if name is not None:
            print('Hello ' + name)
        else:
            print('Hello ')


# Create instance
obj = Human()

# Call the method
obj.sayHello()

# Call the method with a parameter
obj.sayHello('Rover')
```

In method overriding a subclass or child class is able to implement a method that has already been provided by the parent or superclass. The implementation in the child class overrides the implementation of the method in the parent class. In Python this is done by defining a method in the child class with the same name, same parameters and same return type as the method defined in the parent class. If an object of the child class is used to call the method then the child class method is implemented. If an object of the parent class is used to call the method then the parent class method is implemented. In this way the concepts of polymorphism and inheritance are strongly interlinked, as the concept of method overriding is also very important in inheritance.

E.g.

Here the function has the same name but is different according to the class which it is in.

```
class Animal:
   def print_details(self):
       print("This is parent Animal class method")


class Dog(Animal):
   def print_details(self):
       print("This is child Dog class method")


class Cat(Animal):
   def print_details(self):
       print("This is child Cat class method")


rover = Animal()
rover. print_details() #calls the parent Animal method

buster = Dog()
buster.print_details() #calls the child Dog method

fifi = Cat()
fifi.print_details() #calls the child Cat method
```

What is inheritance in OOP?

Inheritance is another one of the four fundamental concepts of OOP. It enables programmers to take all the functionality of one class (the parent or base class) and use it in another class (the child or derived class). Moreover, extra and/or modified functionality can be added to the child class. That is, along with its inherited properties and methods a child class can have its own properties and methods. The concept of inheritance supports code re-usability. Rather than creating new code when we wish to add a new class we can derive our new class from code that already exists.

Diagram - an example of inheritance:

Animal class

Cat class Cow class Dog class

Each class has attributes and methods that are common to the animal class eg. move(), eat(). In addition each child class has attributes and methods that are peculiar to itself e.g. cats climb, dogs swim

The syntax for inheritance in Python looks like this:

```
class Cat(Animal):
   def __init__(self):
       self.character = "independent"
   def kill_mouse(self):
       print("I enjoy hunting and killing mice")

class Dog(Animal):
   def __init__(self):
       self.character = "loyal"
   def fetch_stick(self):
       print("I like to run and collect sticks and bring them back
to my owner")

rover = Dog()
fifi = Cat()

rover.move() #calls the method defined in the parent class Animal
rover.fetch_stick() # calls the method defined in the child class
Dog
print(rover.character) # prints attribute loyal defined in the child
class Dog

fifi.eat()  #calls the method defined in the parent class Animal
fifi.kill_mouse() # calls the method defined in the child class Cat
print(fifi.character) # prints attribute independent defined in the
child class Cat
```

In Python we can use the built-in function isinstance() to check if an object is an instance of a particular class.

```
print(isinstance(rover, Dog))
print(isinstance(rover, Animal))
```

Output is True and True

Similarly we can use issubclass() to check if a class is a child class of another class by passing the child class as the first argument and the parent class as the second argument.

E.g.

```
print(issubclass(Dog, Animal))
Output is True
```

There are different types of inheritance:

- Single - when a child class inherits only one single parent class.
- Multiple - when a child class inherits more than one parent class.
- Multilevel - when a child class is created from another child class.
- Hierarchical - when more than one child class is created from a single parent class.

If you had to make a program that could vote for the top three funniest people in the office, how would you do that? How would you make it possible to vote on those people?

I would create the program using object oriented programming, using classes and objects.

I would use a MySQL database to store the voting options (i.e. who can voters vote for?) and to store the number of votes against each person in the office. There is also the option of storing the voters' information, depending on whether the votes will be anonymous.

I would then create a class vote which would incorporate several methods:

- Get list of people to vote for
- Vote for a person - print("Who would you like to vote for?")
- Shows the total number of votes in the poll

The above methods would be public. There would be other methods, such as taking data from the database and updating the database and these would be protected so that these methods can't be called by the voters i.e the users of the program.

Aspects of the program I would think about would be:

- Would people be able to cast a vote for themselves?
- How would I ensure that each person could only vote once?
- Would the vote include a preference system e.g. the first preference, followed by a second preference and then a third.
- What interface would the voters use to cast their vote? Would an API between a database and a web browser be the best solution?

What is the software development cycle?

The software development life cycle is a methodology with defined processes for creating software. In other words it is the name for the overall process of developing software from beginning to end. The aim of having such a methodology is so that software is produced to the highest quality at the lowest cost in the shortest time possible. When a project is begun it is important to plan how the software will be developed and how it is going to be maintained. There are various theories on the best ways in which this can be done. The reason that this is called a cycle is that with software very often it is necessary to return to the start and change software depending on other technologies which have become available or features which a competitor has developed which now need to be added in order for the product to remain competitive. A software feature can never fully be said to be 'completed' - it will always need updates and maintenance. For example, apps on smartphones are always going to need changes so that they work on the latest smartphones.

The methodology focuses on the following six key stages or phases of software development:

Requirements - also known as the analysis stage. This is when the development team work out what the software needs to do i.e. What are the current problems and what problem is the software going to fix? The main point is to think about what the user will want or need from the program. This involves asking lots of questions of key stakeholders e.g. customers and salespeople. The team also works out the cost and resources required for the implementation of the cycle and assesses feasibility.

Architectural Design - this asks the question "How will we get what we want?" and is when the team work out the details of the program by breaking it down into smaller chunks. This includes thinking about the visual appearance and the programming behind the software. The team will use pseudocode and diagrams to work out how the program should go.

Implementation/Build/Coding - the program code is written. Good pseudocode allows the implementation stage to be relatively easy. The code is normally written in a high-level language.

Testing - this involves testing the program under various conditions to make sure it is going to work. Does the code meet the requirements? The testing looks for defects and deficiencies and fixes any that are found. The team thinks about e.g. what devices the software could be used on and what might cause the program to crash, including edge cases.

Deployment - the software is deployed into production.

Maintenance/Evolution - after the software has been launched the team needs to maintain the product. Software needs to be maintained and to evolve to ensure it works on new systems e.g. smartphone apps are constantly being maintained and changed to make sure they work on the latest smartphones and computers.

What's the difference between Agile and Waterfall?

The Waterfall model of the software development cycle is also known as the linear sequential model. It was created in the 1970s and focuses on an organised and step by step approach to project management i.e. the process moves from one stage to another and each stage must be completed before the next one can be started. It requires key milestones and explanations from clients to the development team as the method focuses on receiving clear requirements from the client in order to accommodate their needs.

In the Waterfall model the different phases are carried out strictly in the following order:-

1. Requirement analysis
2. System Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

The Agile method is a different approach to software development. This method was developed in the 1990s and, as its name suggests, focuses on adaptability rather than meeting strict requirements at each phase before moving forward. It allows development teams to work flexibly.

In the Agile model requirements are agreed on and a basic working product is created quickly which is then revised and updated several times. Great emphasis is placed on client and user feedback which is used to refine and perfect the product. In the agile model work is broken down into smaller incremental builds and done quickly in 'sprints' which last a fixed number of days or weeks in order to get a working product by the end of that time period. At the beginning of each sprint the team meets with the client to outline the goals of that particular sprint. They then develop and test code as they go along.

Both methods have advantages and disadvantages:

Waterfall method - advantages

- Easy to follow and simple to understand
- Relatively easy to manage because each phase is so clearly defined.

Waterfall method - disadvantages

- Can be time consuming - each phase must be completed before the team can move forward. It may not suit short-term projects.
- Inflexible and not adaptable - for example it doesn't work in a project setting in which the requirements of the client are not clearly defined.

Agile method - advantages

- Highly flexible - allows the team to make changes to requirements if/when needed.
- Client satisfaction can be higher- as the development team and the client are in constant communication and client feedback is paramount the client may be overall much happier with the process.
- Features are added quickly - as sprints are short (usually 2-4 weeks) new features can be added rapidly.
- Good for remote teams - can help solve difficulties due to different time zones, communication and availability.

Agile method - disadvantages

- Heavy emphasis on client/customer feedback can sometimes lead the project in the wrong direction.
- Lack of documentation - as the focus is on software quality rather than documenting the process.
- Requires experienced developers - the role of the agile developer is broader than that of the developer in a more traditional model and he/she has responsibilities which go beyond coding e.g. they need to understand client/user needs and take more responsibility for planning and managing tasks.

What is a reduced function used for?

The reduce() function in Python can be found in the functools module.

This function accepts two arguments - a function and an iterable or array - and then returns a single value using the mathematical concept of folding or reduction (reducing or folding a list of values to one single cumulative value). The function which is called by the reduce() function must accept two arguments.

The reduce() function works in the following way. Initially it calls the function using the first two values in the array and returns a result. It then uses that result value and the next value in the array to return another result. It repeats this process recursively until there are no more values left in the array.

E.g.

```
from functools import reduce

def my_sum(a,b):
   result = a + b
   print("{} + {} = {}".format(a,b,result)) #this print shows that
the reduce() function calls the my_sum() function recursively
```

```
    return result

numbers = [1, 2, 3, 4, 5]
print(reduce(my_sum, numbers)) # the final value of the reduce()
function

# Output is:

1 + 2 = 3
3 + 3 = 6
6 + 4 = 10
10 + 5 = 15
15
```

The reduce() function also accepts an optional third argument called the initialiser. If you supply a value to initialiser reduce() will feed it to the first call of function as its first argument. When the function is called it will use the value of the initialiser and the first value in the array to perform its computation. The following example has an initialiser:

```
from functools import reduce

def my_sum(a,b):
    result = a + b
    print("{} + {} = {}".format(a,b,result)) #this print shows that
the reduce() function calls the my_sum() function recursively
    return result

numbers = [1, 2, 3, 4, 5]
print(reduce(my_sum, numbers, 100)) # the final value of the
reduce() function

#Output of the code above is:

100 + 1 = 101
101 + 2 = 103
103 + 3 = 106
106 + 4 = 110
110 + 5 = 115
115
```

Note that it is good practice to use an initialiser when an empty array may potentially be passed as an argument to the function as reduce() will use this as a default return value when an array is empty (rather than raising a type error).

Why is the reduce() function used and when is it useful?

- It can be used to calculate the sum of an array rather than using a for loop (in this way it is similar to the accumulate() function which can be imported from the itertools module).
- It can be used to calculate the product of all the values in an array (N.B. the first value in the sequence must be greater than 0 for this function to work).

How does merge sort work?

Merge sort is a divide and conquer algorithm. This means that a problem is divided into multiple sub-problems and these sub-problems are divided into further sub-problems using the same algorithm until they are simple to solve. Once the sub-problems have been solved they are then combined together to solve the original problem. So, the problem is broken down, each sub-problem is solved recursively and then the solutions are merged in sorted order.

The pseudo code for the algorithm itself is this:

1. Split the array in half
2. Sort the left half of the array
3. Sort the right half of the array
4. Serge the two sorted halves together
5. Steps 1-4 are called recursively through the array

When we are coding we need two separate parts to perform a merge sort. First, code which recursively splits the array in half and second code which merges both halves and returns a sorted array. A merge sort in Python looks like this:

```
def merge_sort(arr):
   if len(arr) > 1:
       left_arr = arr[:len(arr)//2]
       right_arr = arr[len(arr)//2:]


       #recursion
       merge_sort(left_arr)
       merge_sort(right_arr)

       i = 0 #left_arr index
       j = 0 #right arr index
       k = 0 # merged arr index

       while i < len(left_arr) and j < len(right_arr):
           if left_arr[i] < right_arr[j]:
               arr[k] = left_arr[i]
               i += 1
           else:
```

```
                arr[k] = right_arr[j]
                j += 1
            k += 1

        while i < len(left_arr):
            arr[k] = left_arr[i]
            i += 1
            k += 1

        while j < len(right_arr):
            arr[k] = right_arr[j]
            j += 1
            k += 1

arr_test = [7,2,10,0,25,90,2,2,6,0]
merge_sort(arr_test)
print(arr_test)
```

Merge sort is a very efficient sorting algorithm. It has a (O * log(n)) running time which is an optimal running time for comparison based algorithms.

Generator functions allow you to declare a function that behaves like an iterator i.e. it can be used in a for loop. What is the use case?

The concept of generators was introduced in PEP 255 (PEP stands for Python Enhancement Proposals) along with a new statement which is used in conjunction with it - the yield statement. Generator functions are a special type of function which return lazy iterators. Lazy iterators are objects we can loop over - like a list, or dictionary - to access values one at a time except that unlike regular iterators they do not store their data in memory. This means they don't compute the values of each item when they are created, they only compute the values when they are needed. This is known as lazy evaluation and it can drastically reduce the runtime of functions. It is also memory efficient as it takes up less space in the computer.

Generator functions are created in the regular way that functions are created (using the def keyword) except they do not use the return statement. Instead they require the yield keyword, the defining characteristic of a generator function. When we use 'yield' the function automatically becomes a generator function. Generator functions are run by calling the next() function (not by using their name, as with regular functions). Each time the next() function is called the generator produces an iterable one at a time using the yield keyword.

Below is an example of how to create and call a generator function:

```
def my_gen():

   yield 10
```

```
    yield 20

    yield 30

    yield 40


g = my_gen()
print(next(g))
print(next(g))
print(next(g))
print(next(g))
```

```
Output is:

10
20
30
40
```

As is clear above each time the next() function is called the incremented value produced by the yield keyword is printed. This is because, unlike when we use return, the function isn't terminated entirely. Instead the yield keyword saves the state of the function i.e. which value was produced and which is the next one. If you were to call the next() function on the function above once more a StopIteration error would occur. This is because there are no further yield statements in the function to produce another value and the function has therefore been terminated.

Generator expressions, like list comprehensions, are a shorthand way to create generators without using the yield keyword. However, instead of using square brackets [], generator expressions use the () parenthesis. In this way you use a generator without calling a function. E.g.

```
generator = (x ** 2 for x in range(5))

for i in generator:
    print(i)
```

```
Output is:

0
1
4
9
16
```

There are various use cases for generator functions:

- As generators are an excellent way to optimise memory they are often used to iterate through a very large file or data set. For example, some files are larger than the memory in your computer. How will you iterate over its contents if you have to open it and store its contents in a variable? This is where lazy iterators come in handy! Instead of returning each row in a file a generator function opens the file, loops through it and yields it.

- We can use them to either create an infinite sequence or a very large sequence of numbers from 1 to n. This is because generators would iterate over the numbers without storing them in the computer memory, whereas creating a list of large numbers may occupy the entire computer memory (and in the case of an infinite sequence would definitely take up the entire computer memory).

Decorators - A page for useful (or potentially abusive?) decorator ideas. What is the return type of the decorator?

Decorators are a type of design pattern. Design patterns are used by software engineers to create software. The design patterns are a general solution to a problem which occurs often in a given context. Decorators are used to modify the behaviour of function or class. A decorator takes another function as an argument and returns a modified version of it, enhancing its functionality in some way (this is also known as metaprogramming because one part of the program is changing another part of the program at runtime). Decorators are possible in Python because functions are first class objects. This means that they can be used or passed as arguments. decorators are used when we need to modify the behaviour of a function without modifying the function itself.

Like any other function fhe decorator must be defined first before it is used. Here is how we define a decorator function:

```
def my_decorator_func(func):

    def wrapper_func():

        print("Hi")

        func()

        print("Great to have you here!")

    return wrapper_func
```

Once we have defined the decorator function in Python we use the @ symbol as a simplified way of calling it (this is known as syntactic sugar) The decorator is put on the line above the definition of the function which it is going to modify. This syntax makes my_decorator_func() automatically take hello_decorator() as an argument and processes it in its body. E.g

```
@my_decorator_func
def hello_decorator():
```

```
    print("Rachel!")

'''This is equivalent to -

def hello_decorator():
    print("Rachel!")

hello_decorator = my_decorator_function(hello_decorator)'''

The output of hello_decorator() is:

Hi
Rachel!
Great to have you here!
```

In the example above the my_decorator_function is a callable function which, when used, adds some code on top of the hello_decorator function. N.B The return type of the decorator function is the wrapper function. This is the term for the function inside the decorator function I.e. the decorator always returns the inner function inside it.

Below are a few examples of useful decorator ideas:

- Exception handling decorator - by wrapping a try and except clause in an inner function decorators can be applied to different functions to handle exceptions.
- Timing decorator - this times how long it takes a function to run.
- Logging decorator - logs and prints the times at which a function is called and executed.
- Caching decorator - aka memoizing decorator, this caches the results of a function with certain parameters the first time the function is called. After that it returns an answer for the function with a previously provided value without calling the function (as it has already been cached).
- A tracking decorator - counts how many times a function is called.
- Login required decorator - if a user is attempting to view data or perform an action which requires them to be logged in this redirects them to the login URL.
- Permission required decorator - in order to access certain information. A decorator can be used to check whether they have the required permissions.
- Function time-out decorator - if a function is taking too long a timeout and abort function can be installed using a decorator.