Grade: 100 / 100

I could not find a single fault - there was an unbelievable level of depth and critical analysis for each answer, this is a masterpiece of a document. You have truly knocked it out the ball park, amazing work!

THEORY QUESTIONS ASSIGNMENT
Software Stream

Maximum score: 100

KEY NOTES

- This assignment to be completed at student's own pace and submitted before given deadline.
- There are 10 questions in total and each question is marked on a scale 1 to 10. The maximum possible grade for this assignment is 100 points.
- Students are welcome to use any online or written resources to answer these questions.
- The answers need to be explained clearly and illustrated with relevant examples where necessary. Your examples can include code snippets, diagrams or any other evidence-based representation of your answer.

Theory questions	**10 point each**

1. How does Object Oriented Programming differ from Process Oriented Programming?

Object-oriented programming (OOP) is about encapsulating data and behavior into objects. An OOP application will use a collection of objects which knows how to perform certain actions and how to interact with other elements of the application. For example an object could be a person. That person would have a name (that would be a property of the object), and would know how to walk (that would be a method). A method in OOP can be considered as a procedure in PP, but here it belongs to a specific object. Another important aspect of OOP are classes. A class can be considered as a blueprint for an object. ✓ ✓

One reason to use Object-oriented Programming is because it makes it easy to maintain and modify existing code as new objects are created inheriting characteristics from existing ones. This cuts down the development time considerably and makes adjusting the program much simpler. ✓

Another reason to use Object-oriented Programming is the ease of development and ability for other developers to understand the program after development. Well commented objects and classes can tell a developer the process that the developer of the program was trying to follow. It can also make additions to the program much easier for the new developer. ✓

A third to use Object-oriented Programming is the efficiency of the language. Many programming languages using Object-oriented Programming will dump or destroy unused objects or classes freeing up system memory. By doing this the system can run the program faster and more effectively. ✓

Process Orientated Programming

Procedural Orientated Programming, which at times has been referred to as inline programming, takes a more top-down approach to programming. Object-oriented Programming uses classes and objects, Procedural Programming takes on applications by solving problems from the top of the code down to the bottom. It is about writing a list of instructions to tell the computer what to do step by step. It relies on procedures or routines. ✓

This happens when a program starts with a problem and then breaks that problem down into smaller sub-problems or sub-procedures. These sub-procedures are continually broken down in the process called functional decomposition until the sub-procedure is simple enough to be solved. ✓

The issue that is obvious in Procedural Programming is that if an edit is needed to the program, the developer must edit every line of code that corresponds to the original change in the code. An example would be if at the beginning of a program a variable was set to equal the value of 1. If other sub-procedures of the program rely on that variable equaling 1 to function properly they will also need to be edited. As more and more changes may be needed to the code, it becomes increasingly difficult to locate and edit all related elements in the program. ✓

The differences between the two:

Object Oriented Programming	**Procedural Oriented Programming**
In object oriented programming, program is divided into small parts called ***objects***.	In procedural programming, program is divided into small parts called ***functions***. ✓
Object oriented programming follows ***bottom up approach***.	Procedural programming follows ***top down approach***.
Object oriented programming have access specifiers like private, public, protected etc.	There is no access specifier in procedural programming.
Adding new data and function is easy.	Adding new data and function is not easy. ✓
Object oriented programming provides data hiding so it is ***more secure***.	Procedural programming does not have any proper way for hiding data so it is ***less secure***.
Overloading is possible in object oriented programming.	In procedural programming, overloading is not possible. ✓
In object oriented programming, data is more important than function.	In procedural programming, function is more important than data.
Object oriented programming is based on ***real world***.	Procedural programming is based on ***unreal world***.
Examples: C++, Java, Python, C# etc.	Examples: C, FORTRAN, Pascal, Basic etc.

10/10

2. What's polymorphism in OOP?

The literal meaning of polymorphism is the condition of occurrence in different forms.
Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator or object) to represent different types in different scenarios. ✓

Polymorphism allows objects of one type to be treated as objects of another type. For example, the len() function returns the length of the argument passed to it. You can pass a string to len() to see how many characters it has, but you can also pass a list or dictionary to len() to see how many items or key-value pairs it has, respectively. This form of polymorphism is called generic functions or parametric polymorphism, because it can handle objects of many different types.

Polymorphism also refers to ad hoc polymorphism or operator overloading, where operators (such as + or *) can have different behavior based on the type of objects they're operating on. For example, the + operator does mathematical addition when operating on two integer or float values, but it does string concatenation when operating on two strings.

Example 1: Polymorphism in addition operator

We know that the `+` operator is used extensively in Python programs. But, it does not have a single usage. For integer data types, `+` operator is used to perform arithmetic addition operation.

```
num1 = 1
num2 = 2
print(num1+num2)
```

Hence, the above program outputs `3`.
Similarly, for string data types, `+` operator is used to perform concatenation.

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

As a result, the above program output `Python Programming`.

Here, we can see that a single operator `+` has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of polymorphism in Python.

Function Polymorphism in Python

There are some functions in Python which are compatible to run with multiple data types. One such function is the `len()` function. It can run with many data types in Python. Let's look at some example use cases of the function.

Example 2: Polymorphic len() function

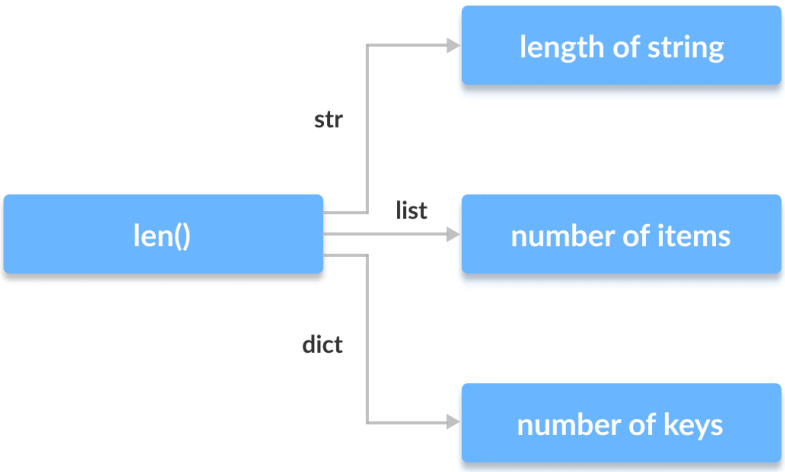
```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Output:

```
9
3
2
```

Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the len() function. However, we can see that it returns specific information about specific data types.

Polymorphism in len() function in Python:



Class Polymorphism in Python

Polymorphism is a very important concept in Object-Oriented Programming. We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name. We can then later generalize calling these methods by disregarding the object we are working with.

Example 3: Polymorphism in Class Methods

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

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years
old.")

    def make_sound(self):
        print("Meow")


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

    def info(self):
```

```
        print(f"I am a dog. My name is {self.name}. I am {self.age} years
old.")

    def make_sound(self):
        print("Bark")


cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```

Output:

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Here, we have created two classes `Cat` and `Dog`. They share a similar structure and have the same method names `info()` and `make_sound()`.

However, notice that we have not created a common superclass or linked the classes together in any way. Even then, we can pack these two different objects into a tuple and iterate through it using a common `animal` variable. It is possible due to polymorphism.

Polymorphism and Inheritance

Like in other programming languages, the child classes in Python also inherit methods and attributes from the parent class. We can redefine certain methods and attributes specifically to fit the child class, which is known as **Method Overriding**.

Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class.

Example 4: Method Overriding

```
from math import pi

class Shape:
    def __init__(self, name):
        self.name = name

    def area(self):
```

```
        pass

    def fact(self):
        return "I am a two-dimensional shape."

    def __str__(self):
        return self.name

class Square(Shape):
    def __init__(self, length):
        super().__init__("Square")
        self.length = length

    def area(self):
        return self.length**2

    def fact(self):
        return "Squares have each angle equal to 90 degrees."

class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def area(self):
        return pi*self.radius**2


a = Square(4)
b = Circle(7)
print(b)
print(b.fact())
print(a.fact())
print(b.area())
```

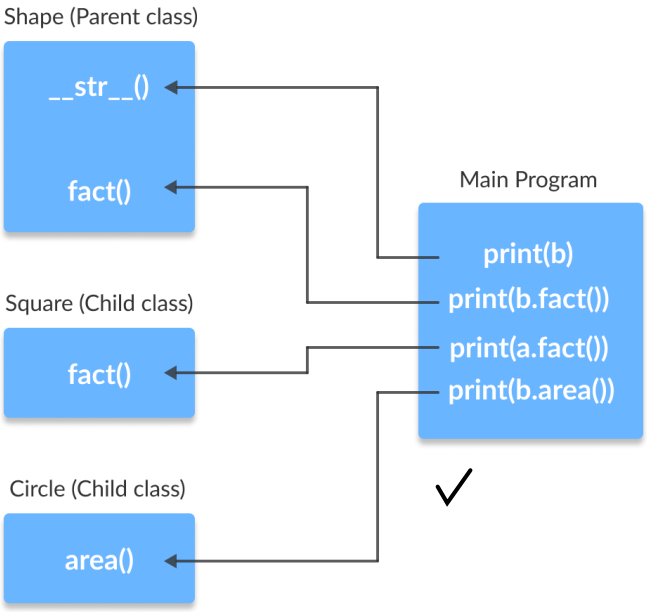
Output:

```
Circle
I am a two-dimensional shape.
Squares have each angle equal to 90 degrees.
153.93804002589985
```

Here, we can see that the methods such as `__str__()`, which have not been overridden in the child classes, are used from the parent class. Due to polymorphism, the Python interpreter automatically recognizes that the `fact()` method for object `a`(`Square` class) is overridden. So, it uses the one

defined in the child class. On the other hand, since the `fact()` method for object `b` isn't overridden, it is used from the Parent `Shape` class.

Polymorphism in parent and child classes in Python:



10/10

Note: **Method Overloading**, a way to create multiple methods with the same name but different arguments, is not possible in Python.

technically true but there are ways to accomplish it tho somewhat as heads up!

3. What's inheritance in OOP?

Inheritance is a code reuse technique that you can apply to classes. It's the act of putting classes into parent-child relationships in which the child class inherits a copy of the parent class's methods, freeing you from duplicating a method in multiple classes.

consider diamond problem as a risk too!

Inheritance is easy to overuse, so can be disliked by programmers and deemed 'dangerous' because of the added complexity that large webs of inherited classes add to a program. But limited use of this technique can be a huge time-saver when it comes to organizing your code. The primary downside of inheritance is that any future changes you make to parent classes are necessarily inherited by all its child classes. In most cases, this tight coupling is exactly what you want. But in some instances, your code requirements won't easily fit your inheritance model.

How Inheritance Works:

```
class ParentClass:
    def printHello(self):
      print('Hello, world!')

class ChildClass(ParentClass):
```

10/10

```
    def someNewMethod(self):
        print('ParentClass objects don't have this method.')

class GrandchildClass(ChildClass):
    def anotherNewMethod(self):
        print('Only GrandchildClass objects have this method.')

print('Create a ParentClass object and call its methods:')
parent = ParentClass()
parent.printHello()

print('Create a ChildClass object and call its methods:')
child = ChildClass()
child.printHello()
child.someNewMethod()

print('Create a GrandchildClass object and call its methods:')
grandchild = GrandchildClass()
grandchild.printHello()
grandchild.someNewMethod()
grandchild.anotherNewMethod()
```

When you run this program, the output should look like this:

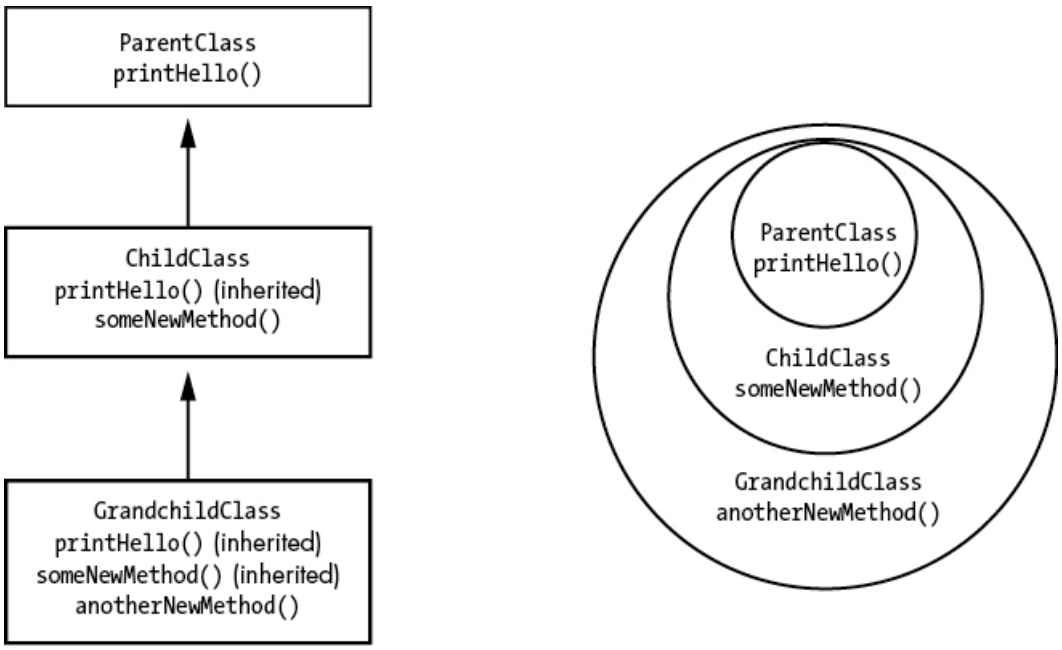
```
Create a ParentClass object and call its methods:
Hello, world!
Create a ChildClass object and call its methods:
Hello, world!
ParentClass objects don't have this method.
Create a GrandchildClass object and call its methods:
Hello, world!
ParentClass objects don't have this method.
Only GrandchildClass objects have this method.
```

We've created three classes named `ParentClass 1`, `ChildClass 3`, and `GrandchildClass 4`. The `ChildClass` ***subclasses*** `ParentClass`, meaning that `ChildClass` will have all the same methods as `ParentClass`. We say that `ChildClass` ***inherits*** methods from `ParentClass`. Also, `GrandchildClass` subclasses `ChildClass`, so it has all the same methods as `ChildClass` and its parent, ParentClass.

Using this technique, we've effectively copied and pasted the code for the printHello() method 2 into the ChildClass and GrandchildClass classes. Any changes we make to the code in printHello() update not only ParentClass, but also ChildClass and GrandchildClass. This is the same as changing the code in a function updates all of its function calls.

You can see this relationship in the image below. Notice that in class diagrams, the arrow is drawn from the subclass pointing to the base class. This reflects the fact that a class will always know its base class but won't know its subclasses.

A hierarchical diagram (left) and Venn diagram (right) showing the relationships between the three classes and the methods they have:



It's common to say that parent-child classes represent "is a" relationships. A ChildClass object is a ParentClass object because it has all the same methods that a ParentClass object has, including some additional methods it defines. This relationship is one way: a ParentClass object is not a ChildClass object. If a ParentClass object tries to call someNewMethod(), which only exists for ChildClass objects (and the subclasses of ChildClass), Python raises an AttributeError.

Programmers often think of related classes as having to fit into some real-world "is a" hierarchy. OOP tutorials commonly have parent, child, and grandchild classes of:

Vehicle ▶ FourWheelVehicle ▶ Car
Animal ▶ Bird ▶ Sparrow
Shape ▶ Rectangle ▶ Square

Remember: the primary purpose of inheritance is code reuse. If your program needs a class with a set of methods that is a complete superset of some other class's methods, inheritance allows you to avoid copying and pasting code.

We also sometimes call a child class a subclass or derived class and call a parent class the super class or base class.

4. 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?

Approach:
1. Begin with an input statement to allow everyone to vote for their chosen person ✓
2. Convert the votes to all lower case and place into an array of names of the candidates.
3. A candidate name in the array represents a vote cast to the candidate. Print the name of candidates who received Max vote.
4. If there is tie, print a lexicographically smaller name. ✓ good consideration of edge case!

For example:

Input : votes[] = {"john", "johnny", "jackie",
"johnny", "john", "jackie",
"jamie", "jamie", "john",
"johnny", "jamie", "johnny",
"john"};

10/10

what if people have the same name in the office? how do we distinguish?

Output : John

We have four Candidates with name as 'John', 'Johnny', 'jamie', 'jackie'. The candidates John and Johny get maximum votes. Since John is alphabetically smaller, we print it.

Method:
This is a shorter method using libraries, but makes things so much easier: ✓
1. Count the number of votes for each person and stores in a dictionary.
2. Find the maximum number of votes.
3. Find corresponding person(s) having votes equal to maximum votes.
4. As we want output according to lexicographical order, so sort the list and print first element.

Code:

```
from collections import Counter

#Get votes through an input question:
vote_choice = input("Who do you think is the funniest person in the office?")

#Format answer
answer = vote_choice.lower().strip()

#convert answer into an array:
votes =['john','johnny','jackie','johnny','john','jackie',
    'jamie','jamie','john','johnny','jamie','johnny','john']

#Count the votes for persons and stores in the dictionary
vote_count=Counter(votes)

#Find the maximum number of votes
max_votes=max(vote_count.values())

#Search for people having maximum votes and store in a list
lst=[i for i in vote_count.keys() if vote_count[i]==max_votes]

#Sort the list and print lexicographical smallest name
print(sorted(lst)[0])
```

✓ ✓

Output:

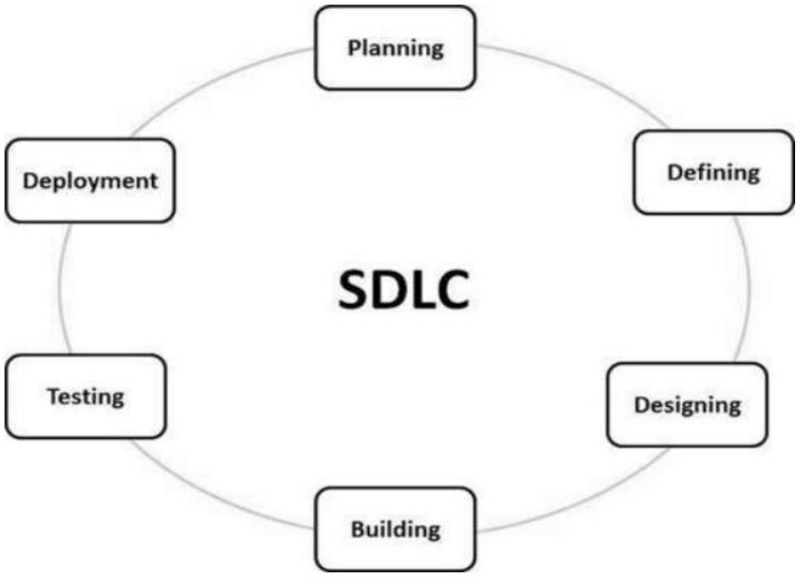
```
John
```

5. What's the software development cycle?

Software Development Life Cycle (SDLC) is a process used by the software industry to design, develop and test high quality softwares. The SDLC aims to produce a high-quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates. It is a framework defining tasks performed at each step in the software development process. ISO/IEC 12207 is an international standard for software life-cycle processes. It aims to be the standard that defines all the tasks required for developing and maintaining software.

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

The following figure is a graphical representation of the various stages of a typical SDLC.



A typical Software Development Life Cycle consists of the following stages −

Stage 1: Planning and Requirement Analysis
Requirement analysis is the most important and fundamental stage in SDLC. It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry. This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas.

Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage. The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks.

Stage 2: Defining Requirements
Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts. This is done through an SRS (Software Requirement Specification) document which consists of all the product requirements to be designed and developed during the project life cycle.

Stage 3: Designing the Product Architecture
SRS is the reference for product architects to come out with the best architecture for the product to be developed. Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification.

This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product.

A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any). The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS.

Stage 4: Building or Developing the Product
In this stage of SDLC the actual development starts and the product is built. The programming code is generated as per DDS during this stage. If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle.

Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code. Different high level programming languages such as C, C++, Pascal, Java and PHP are used for coding. The programming language is chosen with respect to the type of software being developed.

Stage 5: Testing the Product
This stage is usually a subset of all the stages as in the modern SDLC models, the testing activities are mostly involved in all the stages of SDLC. However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS.

Stage 6: Deployment in the Market and Maintenance
Once the product is tested and ready to be deployed it is released formally in the appropriate market. Sometimes product deployment happens in stages as per the business strategy of that organization. The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing).

Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment. After the product is released in the market, its maintenance is done for the existing customer base.

SDLC Models
There are various software development life cycle models defined and designed which are followed during the software development process. These models are also referred as Software Development

Process Models". Each process model follows a Series of steps unique to its type to ensure success in the process of software development.

The following are the most important and popular SDLC models followed in the industry:

- Waterfall Model
- Iterative Model
- Spiral Model
- V-Model
- Big Bang Model

Other related methodologies are Agile Model, RAD Model, Rapid Application Development and Prototyping Models.

10/10

6. What's the difference between agile and waterfall?

In the ever-expanding world of project management methodologies two of the heaviest hitters are Agile and Waterfall.

Both have been around for some time, with Waterfall appearing some 50 years ago and Agile, a relative newcomer, at about 20 years old. Both emerged as tools for principally managing software development but are now widely applied across very different types of projects.

What is the difference?

To put it simply Waterfall is essentially making a good plan and sticking to it, while Agile utilises a more flexible, iterative approach. Waterfall is more sequential and pre-defined, while Agile is more adaptable as a project progresses.

Agile is more a set of principles than one methodology. Its principles are applied in other more specific methodologies such as Scrum, eXtreme Programming, Kanban, and Scrumban. It is a practice based on continuous iterations of development and testing where such activities can run concurrently. Agile projects are characterised by a series of tasks that are conceived, executed and adapted as the situation demands, rather than a pre-planned process.

Waterfall, on the other hand, is much more linear, focusing on up front planning with requirements fully defined before a project commences. Like its name suggests, work cascades, much like a waterfall, through different project phases. Each phase needs to be completed before the next one can begin.

Pros and Cons

Agile pros:

Agile's flexibility avoids rigidity. Stakeholders and team members have opportunities to observe and test throughout the project which allows for adjustments and changes to be made as things move forward. This greater "user focus" means that on delivery it's likely the outcome will be more in line with expectations - even if they have evolved along the way.

Agile encourages teamwork, collaboration, self-organisation and accountability. This helps with overall motivation and commitment to a project's outcomes and goals.

Agile cons:
Because of its flexibility and ability to adapt, an Agile project can run the risk of losing direction or falling behind schedule. Constantly iterating and re-aligning a project can easily stretch budgets and timelines. ✓

Adherence to Agile principles mean that team members are almost completely dedicated to the project which can be challenging from a resource or continuity point of view. While this kind of focus can lead to faster outcomes it may not lead to better ones if the project feels uncontained or has lost focus. ✓

Waterfall pros:
Waterfall is particularly efficient for well-defined projects. Project stakeholders agree upfront on what will be delivered, which makes planning and design much easier. Progress is more easily tracked as the full scope of the project is known from the beginning. ✓

Unlike Agile, this more linear approach often means that team members only need to be available for their specific project phases and can thus continue to focus in other areas. Equally a project's customers may only need to be involved heavily in the early initial scoping phase and then at delivery. ✓

Waterfall cons:
Waterfall requires comprehensive requirements up front which can sometimes be challenging for more complex or longer-term projects.

Its sequential nature and reliance on pre-planning means there is a certain rigidity built into the project that makes mid-project pivots or directional shifts difficult to deploy without re-engineering all those pre-made plans.

How to choose?
As always, choosing which methodology (Agile or Waterfall) is right for your project is entirely dependent on the nature of the project you have in mind and the culture and type of organisation you work for. Of course, the reality might be that a hybrid Agile Waterfall approach makes the most sense for your project but there are some general rules of thumb. ✓

Agile
Agile makes sense where a project is based on incremental progress, complex deliverables or consists of multiple, not always sequential timelines. Projects that require cohesive and collaborative but cross-functional teams to deliver will need to take an Agile approach. ✓

If processes or roles are unclear it makes room for figuring it out as the project goes along. It also allows for involving the project client at any stage along the way. Products that are developed in stages, updates or versions are particularly suited to Agile. ✓

^ this section is brilliant critical analysis

Waterfall
Waterfall is usually more suited to less complex projects or those that have well defined requirements, processes and roles for team members. Single delivery timeframes with lots of detail and an expectation that very little is likely to change along the way are ideal. ✓

Waterfall also works well when the client is not required to be heavily involved beyond the initial brief and final delivery. From a management point of view, Waterfall can often make sense for fixed-price or contract dependent projects in order to lessen the risk of budget or delivery over-runs. ✓

7. What is a reduced function used for?

Python's **reduce()** implements a mathematical technique commonly known as **folding** or **reduction**. You're doing a fold or reduction when you reduce a list of items to a single cumulative value. Python's reduce() operates on any iterable—not just lists—and performs the following steps:

1. **Apply** a function (or callable) to the first two items in an iterable and generate a partial result.
2. **Use** that partial result, together with the third item in the iterable, to generate another partial result.
3. **Repeat** the process until the iterable is exhausted and then return a single cumulative value.

Using a For Loop

The best way to introduce the reduce() function is to start with a problem and attempt to solve it the old-fashioned way, using a for loop. Then we can attempt the same task using the reduce function. Let's say that we have a list of numbers and we want to return their product. In other words, we want to multiply all the numbers in the list together and return that single value. We can accomplish this using a for loop:

```
# list of numbers
num_list = [1,2,3,4,5]

# set product equal to 1
product = 1

# loop through num_list and multiply the numbers together
for num in num_list:
    product = product * num

print(product) # output is 120
```

*We have our list of numbers, **num_list**. We want to multiply the numbers in this list together and get their product. We create the variable **product** and set it equal to 1. We then loop through **num_list** using a for loop, and we multiply each number by the result of the previous iteration. After looping through **num_list**, the **product**, or the accumulator, will equal 120, which is the product of all the numbers in the list.*

Reduce Function

It turns out that we can accomplish the above task using the reduce function instead of a for loop. The reduce function can take in three arguments, two of which are required. The two required arguments are: a function (that itself takes in two arguments), and an iterable (such as a list).

```
reduce(function, iterable[, initializer])
```

Importing Reduce

The reduce function is in the module functools, which contains higher-order functions. Higher-order functions are functions that act on or return other functions. Therefore, in order to use the reduce function, we either need to import the entire functools module, or we can import only the reduce function from functools:

```
import functools
from functools import reduce
```
Note: *if we import functools, we would need to access the reduce function like so: functools.reduce(arguments). If we import the reduce function only from the functools module, we can access the reduce function by just typing reduce(arguments).*

Exploring the Reduce Function

As previously noted, the reduce function takes in two required arguments: a function and an iterable. **In order to avoid confusion between the reduce function and the function it takes in as an argument, I will refer to the reduce function as just reduce**.

The first argument that reduce takes in, the function, must itself take in two arguments. Reduce will then apply this function cumulatively to the elements of the iterable (from left to right), and reduces it to a single value.

Example:

Let's say the iterable we use is a list, such as **num_list** from the above example:

```
num_list = [1,2,3,4,5]
```

And the function we use as the first argument for reduce is the following:

```
def prod(x, y):
    return x * y
```
*This prod function takes in two arguments: x and y. It then returns their product, or x * y.*

Let's pass in the **prod** function and **num_list** as our function and iterable, respectively, to reduce:

```
from functools import reduceproduct = reduce(prod, num_list)
```

Our iterable object is ***num_list***, which is the list: [1,2,3,4,5]. Our function, ***prod***, takes in two arguments, x and y. Reduce will start by taking the first two elements of ***num_list***, 1 and 2, and passes them in to our ***prod*** function as the x and y arguments. The ***prod*** function returns their product, or 1 * 2, which equals 2. Reduce will then use this accumulated value of 2 as the new or updated x value, and uses the next element in ***num_list***, which is 3, as our new or updated y value. It then sends these two values (2 and 3) as x and y to our ***prod*** function, which then returns their product, 2 * 3, or 6. This 6 will then be used as our new or updated x value, and the next element in ***num_list*** will be used as our new or updated y value, which is 4. It then sends 6 and 4 as our x and y values to the ***prod*** function, which returns 24. Thus, our new x value is 24, and our new y value is the next element from ***num_list***, or 5. These two values are passed to the ***prod*** function as our x and y values, and ***prod*** returns their product, 24 * 5, which equals 120. Thus, reduce took an iterable object, in this case ***num_list***, and reduced it down to a single value, 120. This value is then assigned to the variable product. ***In other words, the x argument gets updated with the accumulated value, and the y argument gets updated from the iterable.***

Third Argument: Initializer

Remember how we said that reduce can take in an optional third argument, the initializer? The default value for it is None. If we pass in an initializer, it will be used as the first x value by reduce (instead of x being the first element of the iterable). So if we passed in the number 2 in the above example as the initializer:

```
product = reduce(prod, num_list, 2)print(product) #240
```

Then the first two values or arguments for x and y will be 2 and 1, respectively. All subsequent steps will be the same. In other words, the initializer is placed before the elements of our iterable in the calculation.

Other Examples of Using Reduce

There are so many other scenarios of when we can use reduce.

For example, we can find the sum of the numbers in a list:

```
num_list = [1,2,3,4,5]sum = reduce(lambda x,y: x + y, num_list)print(sum) #15
```

Or we can find the maximum number in a list:

```
num_list = [1,2,3,4,5]max_num = reduce(lambda x,y: x if x > y else y, num_list)print(max_num) #5
```

Or the minimum number in a list:

```
num_list = [1,2,3,4,5]min_num = reduce(lambda x,y: x if x < y else y, num_list)print(min_num) #1
```

Note: Python does have built-in functions such as max(), min(), and sum() that would have been easier to use for these three examples, so reduce() can be used to accomplish many different tasks.

10/10

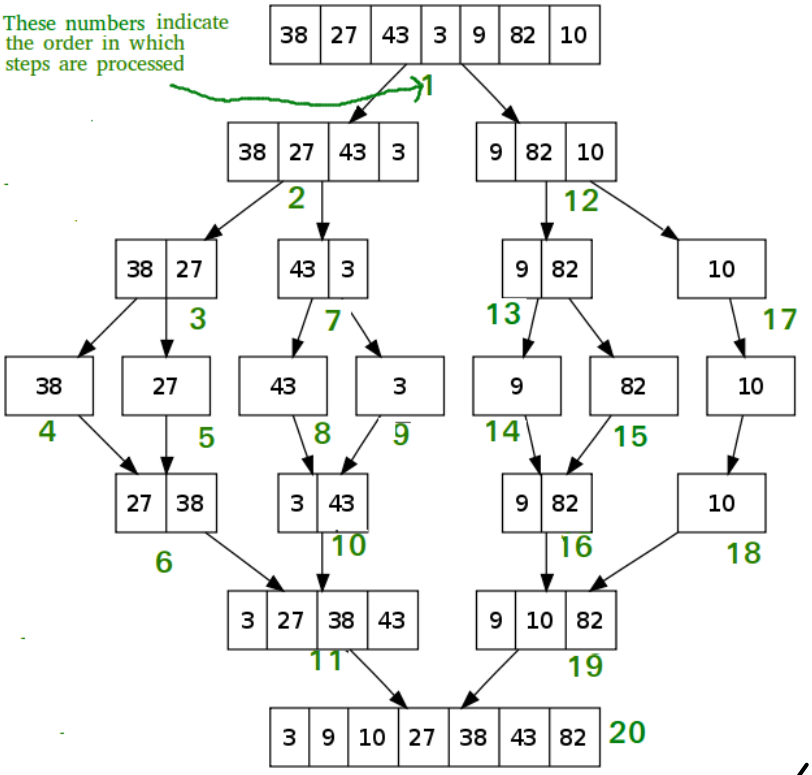
8. How does merge sort work?

Merge Sort is a recursive, 'divide and conquer' algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

The process in action:

Array to be sorted:
[8, 1, 5, 2, 10, 7, 3, 4]
Step 1: divide the list into lists of 1:
[8], [1], [5], [2], [10], [7], [3], [4]
Step 2: compare the list in 2s, sorting those two by lowest first:
[1,8], [2,5], [7,10], [3,4]
Step 3: take the pairs of lists and sort the first element in each to see which is smaller, then place into 2 arrays of 4:
[1,2,5,8], [3,4,7,10]
Step 4: compares the first element of each array, sorting as it goes, landing on the final list:
[1,2,3,4,5,7,8,10]

This can be seen more visually in the diagram below, which shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

should norm be n log n, no?

Auxiliary Space: O(n)

Algorithmic Paradigm: Divide and Conquer

Applications of Merge Sort

- Merge Sort is useful for sorting linked lists in O(nLogn) time. In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore, the merge operation of merge sort can be implemented without extra space for linked lists.
- Inversion Count Problem
- Used in External Sorting

Drawbacks of Merge Sort:

- Slower comparative to the other sort algorithms for smaller tasks.
- Merge sort algorithm requires an additional memory space of 0(n) for the temporary array.
- It goes through the whole process even if the array is sorted.

10/10

9. Generators - 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?

Generator

In Python, a generator is a function that returns an iterable. An iterable is an object capable of producing its members one at a time (lists, tuples, strings, etc.). However, the iterable returned from a generator is **different** in that it is considered a generator object. When we begin iterating over a

generator object, we are not merely reading from memory. Instead, we are executing code in the body of our generator function for each iteration. The most defining feature of a python generator is the “yield” statement. The main difference between yield and return statements is that a yield statement does not terminate the function. A yield statement pauses the operation and remembers the state, and can later continue on further iterations.

Example of a generator function that returns a generator object that yields the input string in reverse order:

```
1   def reverse_str(str):
2       for i in range(len(str)-1, -1, -1):
3           yield str[i]
4
5   for char in reverse_str("hello"):
6       print(char)
7
8   """
9   ->
10  o
11  l
12  l
13  e
14  h
15  """
```

In lines 1–4, we define the generator. In lines 5–6, we iterate over the returned generator object.

Iterator

Iterators versus iterables: an iterable is an object capable of being iterated. An iterator is an object that is used to iterate over an iterable. All iterator objects have a “__next__” method implementation that is used to return the next item in the iterable, as well as raise “StopIteration” to signal that there is no more iteration to be done. The __next__() method will also update the object’s state to point to the next value in the iterable. Iterables and some iterators have an “__iter__” method that is used to return an iterator.

Example of an iterator class that counts to ten:

```python
class count_to_ten:
    def __init__(self):
        self.max = 10

    def __iter__(self):
       self.n = 0
       return self

    def __next__(self):
        if self.n < self.max:
            self.n += 1
            return self.n
        else:
            raise StopIteration


count_obj = count_to_ten()
count_iter = iter(count_obj)

for i in count_iter:
    print(i)

"""
->
1
2
3
4
5
6
7
8
9
10
"""
```

Should I use an iterator or a generator?

A generator is initialized as a function, whereas an iterator requires writing a class. Therefore generators are not only easier to implement but also increase our code's readability. ✓
Iterators are useful when we need to extend any object that needs to be iterated over. Or if we need a class that has complex state management and other methods. Otherwise, for the majority of cases, a generator is suited.

Comparison Between Python Generator vs Iterator

1. The difference between Iterators and Generators in python:
2. In creating a python generator, we use a function. But in creating an iterator in python, we use the iter() and next() functions. ✓
3. A generator in python makes use of the 'yield' keyword. A python iterator doesn't.

4. Python generator saves the states of the local variables every time 'yield' pauses the **loop in python**. An iterator does not make use of local variables, all it needs is iterable to iterate on.
5. A generator may have any number of 'yield' statements.
6. You can implement your own iterator using a **python class**; a generator does not need a class in python.
7. To write a python generator, you can either use a **Python function** or a comprehension. But for an iterator, you must use the iter() and next() functions.
8. Generator in python let us write fast and compact code. This is an advantage over Python iterators. They are also simpler to code than do custom iterator.
9. Python iterator is more memory-efficient.

10/10

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

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure. Decorators are usually called before the definition of a function you want to decorate. In this tutorial, we'll show the reader how they can use decorators in their Python functions.

Functions in Python are first class citizens. This means that they support operations such as being passed as an argument, returned from a function, modified, and assigned to a variable. This is a fundamental concept to understand before we delve into creating Python decorators.

Assigning Functions to Variables

To kick us off we create a function that will add one to a number whenever it is called. We'll then assign the function to a variable and use this variable to call the function.

```
def plus_one(number):
    return number + 1


add_one = plus_one
add_one(5)
```

```
6
```

Defining Functions Inside other Functions

Next, we'll illustrate how you can define a function inside another function in Python. Stay with me, we'll soon find out how all this is relevant in creating and understanding decorators in Python.

```
def plus_one(number):
    def add_one(number):
        return number + 1


    result = add_one(number)
    return result
plus_one(4)
```

```
5
```

Passing Functions as Arguments to other Functions
Functions can also be passed as parameters to other functions.
For Example:

```
def plus_one(number):
    return number + 1


def function_call(function):
    number_to_add = 5
    return function(number_to_add)


function_call(plus_one)
```

```
6
```

Functions Returning other Functions
A function can also generate another function.
Example:

```
def hello_function():
    def say_hi():
        return "Hi"
    return say_hi
hello = hello_function()
hello()
```

Output: 'Hi'

Nested Functions have access to the Enclosing Function's Variable Scope
Python allows a nested function to access the outer scope of the enclosing function. This is a critical concept in decorators -- this pattern is known as a Closure.

```
def print_message(message):
   "Enclosong Function"
   def message_sender():
      "Nested Function"
      print(message)

   message_sender()

print_message("Some random message")
```

Output: Some random message

Creating Decorators

With these prerequisites out of the way, let's go ahead and create a simple decorator that will convert a sentence to uppercase. We do this by defining a wrapper inside an enclosed function. As you can see it very similar to the function inside another function that we created earlier.

```
def uppercase_decorator(function):
   def wrapper():
      func = function()
      make_uppercase = func.upper()
      return make_uppercase

   return wrapper
```

Our decorator function takes a function as an argument, and we shall, therefore, define a function and pass it to our decorator. We learned earlier that we could assign a function to a variable. We'll use that trick to call our decorator function.

```
def say_hi():
   return 'hello there'

decorate = uppercase_decorator(say_hi)
decorate()
```

Output: 'HELLO THERE'

However, Python provides a much easier way for us to apply decorators. We simply use the @ symbol before the function we'd like to decorate.
Example:

```
@uppercase_decorator
def say_hi():
   return 'hello there'

say_hi()
```

Output: 'HELLO THERE'

Applying Multiple Decorators to a Single Function

We can use multiple decorators to a single function. However, the decorators will be applied in the order that we've called them. Below we'll define another decorator that splits the sentence into a list. We'll then apply the **uppercase_decorator** and **split_string** decorator to a single function.

```
def split_string(function):
    def wrapper():
        func = function()
        splitted_string = func.split()
        return splitted_string
    return wrapper

@split_string
@uppercase_decorator
def say_hi():
    return 'hello there'
say_hi()

Output:  ['HELLO', 'THERE']
```

From the above output, we notice that the application of decorators is from the bottom up. Had we interchanged the order, we'd have seen an error since lists don't have an upper attribute. The sentence has first been converted to uppercase and then split into a list.

Accepting Arguments in Decorator Functions

Sometimes we might need to define a decorator that accepts arguments. We achieve this by passing the arguments to the wrapper function. The arguments will then be passed to the function that is being decorated at call time.

```
def decorator_with_arguments(function):
    def wrapper_accepting_arguments(arg1, arg2):
        print("My arguments are: {0}, {1}".format(arg1,arg2))
        function(arg1, arg2)
    return wrapper_accepting_arguments

@decorator_with_arguments
def cities(city_one, city_two):
    print("Cities I love are {0} and {1}".format(city_one, city_two))

cities("Nairobi", "Accra")

Output:
My arguments are: Nairobi, Accra
Cities I love are Nairobi and Accra
```

Defining General Purpose Decorators

To define a general purpose decorator that can be applied to any function we use args and **kwargs. args and **kwargs collect all positional and keyword arguments and stores them in the args and kwargs variables. args and kwargs allow us to pass as many arguments as we would like during function calls.

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
```

```
    def a_wrapper_accepting_arbitrary_arguments(*args,**kwargs):
        print('The positional arguments are', args)
        print('The keyword arguments are', kwargs)
        function_to_decorate(*args)
    return a_wrapper_accepting_arbitrary_arguments

@a_decorator_passing_arbitrary_arguments
def function_with_no_argument():
    print("No arguments here.")

function_with_no_argument()
```

Output:
The positional arguments are ()
The keyword arguments are {}
No arguments here.

Let's see how we'd use the decorator using positional arguments.

```
@a_decorator_passing_arbitrary_arguments
def function_with_arguments(a, b, c):
    print(a, b, c)

function_with_arguments(1,2,3)
```

Output:
The positional arguments are (1, 2, 3)
The keyword arguments are {}
1 2 3

Keyword arguments are passed using keywords.
Example:

```
@a_decorator_passing_arbitrary_arguments
def function_with_keyword_arguments():
    print("This has shown keyword arguments")

function_with_keyword_arguments(first_name="Derrick", last_name="Mwiti")
```

Output:
The positional arguments are ()
The keyword arguments are {'first_name': 'Derrick', 'last_name': 'Mwiti'}
This has shown keyword arguments

Passing Arguments to the Decorator

Now let's see how we'd pass arguments to the decorator itself. In order to achieve this, we define a decorator maker that accepts arguments then define a decorator inside it. We then define a wrapper function inside the decorator as we did earlier.
Example:

```
def decorator_maker_with_arguments(decorator_arg1, decorator_arg2, decorator_arg3):
    def decorator(func):
```

```
        def wrapper(function_arg1, function_arg2, function_arg3) :
            "This is the wrapper function"
            print("The wrapper can access all the variables\n"
                "\t- from the decorator maker: {0} {1} {2}\n"
                "\t- from the function call: {3} {4} {5}\n"
                "and pass them to the decorated function"
                .format(decorator_arg1, decorator_arg2,decorator_arg3,
                        function_arg1, function_arg2,function_arg3))
            return func(function_arg1, function_arg2,function_arg3)

        return wrapper

    return decorator

pandas = "Pandas"
@decorator_maker_with_arguments(pandas, "Numpy","Scikit-learn")
def decorated_function_with_arguments(function_arg1, function_arg2,function_arg3):
    print("This is the decorated function and it only knows about its arguments: {0}"
          " {1}" " {2}".format(function_arg1, function_arg2,function_arg3))

decorated_function_with_arguments(pandas, "Science", "Tools")

Output:
The wrapper can access all the variables
    - from the decorator maker: Pandas Numpy Scikit-learn
    - from the function call: Pandas Science Tools
and pass them to the decorated function
This is the decorated function, and it only knows about its arguments: Pandas Science Tools
```

Debugging Decorators

As we have noticed, decorators wrap functions. The original function name, its docstring, and parameter list are all hidden by the wrapper closure: For example, when we try to access the **decorated_function_with_arguments** metadata, we'll see the wrapper closure's metadata. This presents a challenge when debugging.

```
decorated_function_with_arguments.__name__
Output:  'wrapper'

decorated_function_with_arguments.__doc__
Output:  'This is the wrapper function'
```

In order to solve this challenge Python provides a functools.wraps decorator. This decorator copies the lost metadata from the undecorated function to the decorated closure.
Example:

```
import functools

def uppercase_decorator(func):
    @functools.wraps(func)
    def wrapper():
        return func().upper()
```

```
    return wrapper

@uppercase_decorator
def say_hi():
    "This will say hi"
    return 'hello there'

say_hi()

Output:  'HELLO THERE'
```

When we check the **say_hi** metadata, we notice that it is now referring to the function's metadata and not the wrapper's metadata.

```
say_hi.__name__
Output:  'say_hi'

say_hi.__doc__
Output:  'This will say hi'
```

It is advisable and good practice to always use **functools.wraps** when defining decorators - It saves a lot of headache in debugging.

Python Decorators Summary

Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated. Using decorators in Python also ensures that your code is DRY(Don't Repeat Yourself). Decorators have several use cases such as:

- Authorization in Python frameworks such as Flask and Django
- Logging
- Measuring execution time
- Synchronization