THEORY QUESTIONS ASSIGNMENT

Software Stream

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? .

Process-oriented programming is a paradigm based on Communicating Sequential Processes, originally from a paper by Tony Hoare in 1977. This is also popularly called the actor model of concurrency. Other languages with some relation to this original work include Occam, Limbo, and Go. It has a top-down approach to programming.
In OOP the code is easy to maintain and the blocks can be reused for multiple purpuses.

2. **What's polymorphism in OOP?**

Polymorphism is one of the core concepts of object-oriented programming (OOP) and describes situations in which something occurs in several different forms.
So polymorphism is the ability to present the same interface for differing underlying forms (data types).
One example of polymorphism in OOP is the use of the addition operator.
If using '+' on integer data types, the operator is used to perform arithmetic:

```
x = 5
y = 2
print(x + y)
#output: 7
```

For strings the operator performs concatenation:

```
x = "cats"
y = "are great"
print(x + y)
#output: "cats are great"
```

Polymorphism can also be used in functions:

len() function on different data types (string, list).

```
print(len("Software"))
#output: 8
print(len(["Software", "Data", "Full-Stack"]))
#output: 3
```

Polymorphism in Classes

```
class Data_Student:
    def __init__(self, name, favourite):
        self.name = name
        self.favourite = favourite

    def info(self):
        print(f"I am a Data Student. My name is {self.name}. My favourite topic is

    def specialisation(self):
        print("Machine Learning")


class Software_Student:
    def __init__(self, name, favourite):
        self.name = name
        self.favourite = favourite

    def info(self):
       print(f"I am a Software Student. My name is {self.name}. My favourite topic

    def specialisation(self):
        print("Back-End in Python")

data1 = Data_Student("Ozge", "K-means clustering")
software1 = Software_Student("Alline", "APIs")

for student in (data1, software1):
    student.specialisation()
    student.info()
```

Output: Machine Learning I am a Data Student. My name is Ozge. My favourite topic is in K-means clustering Back-End in Python I am a Software Student. My name is Alline. My favourite topic in APIs

Here, we have created two classes Data_Student and Software_Student. They share a similar structure and have the same method names info() and specialisation().

without a common superclass or linking the classes together these two different objects can be put into a

tuple and iterate through it using a common *student* variable. It is possible due to polymorphism.

3. What's inheritance in OOP?

Inheritance is a concept in OOP where a class can inherit all methods and properties from another class. In inheritance, there is a parent class (aka “super class” or “base class”) and a child class (aka “subclass” or “derived class”).

advantages of inheritance include:

- Reusability of code
- No data duplications and redundant data
- Reduces space and time complexity
- Helps improve structure of code

disadvantages of inheritance include:

- Can slow down the execution of code
- Problems can arise from parent and child classes being tightly coupled.

```
class Student:

    def __init__(self, name, age, id):
        self.name = name

        self.age = age

        self.id = id

#child class

class Software_Student(Student):

    def __init__(self, name, age, id, pathway, grades):
        super().__init__(name, age, id)
        self.pathway = pathway
        self.grades = grades


class Fullstack_Student(Student):

    def __init__(self, name, age, id, pathway, grades):
        super().__init__(name, age, id)
        self.pathway = pathway
        self.grades = grades
        self.student_grades = {}
```

When we add a child class (a stream student) the__ init__ parameters can overwrite the parent class(Student) it is important to add super().__init__ to inherit from Student.

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?

1. ask all participants to vote for the one they think is the funniest person in the office.: They would

enter one person's name in response to the prompt. Every participant can only vote once.
2. append every input into the list.
3. use the Counter method from collections, to create a dictionary of keys and values. The keys would be the chosen collegues and the values would be the total of all the votes they got.
4. create a new list, with only the keys of the dictionary
5. sort new dictionary in reverse order with sorted() function.
6. print the first 3 items in the new list

5. **What's the software development cycle?**

Software Development Life Cycle (SDLC) is the application of standard business practices to building software applications. It's typically divided into six to eight steps: Planning, Analysis, Designing, Building, (Documenting), Testingt,(/ Deploying), Maintainining

A closer look:

1. Planning and requirement gathering - Inputs from user and general all stakeholders and other relevant parties are taken on board to establish product requirements. This forms the basis of the basic product design.

2.Analysing - All requirements specified initially are analysed and finalised into in this stage. The requirements then receive approval from relevant parties. A Software Requirement Specification (SRS) document is compiled outlining those specifications.

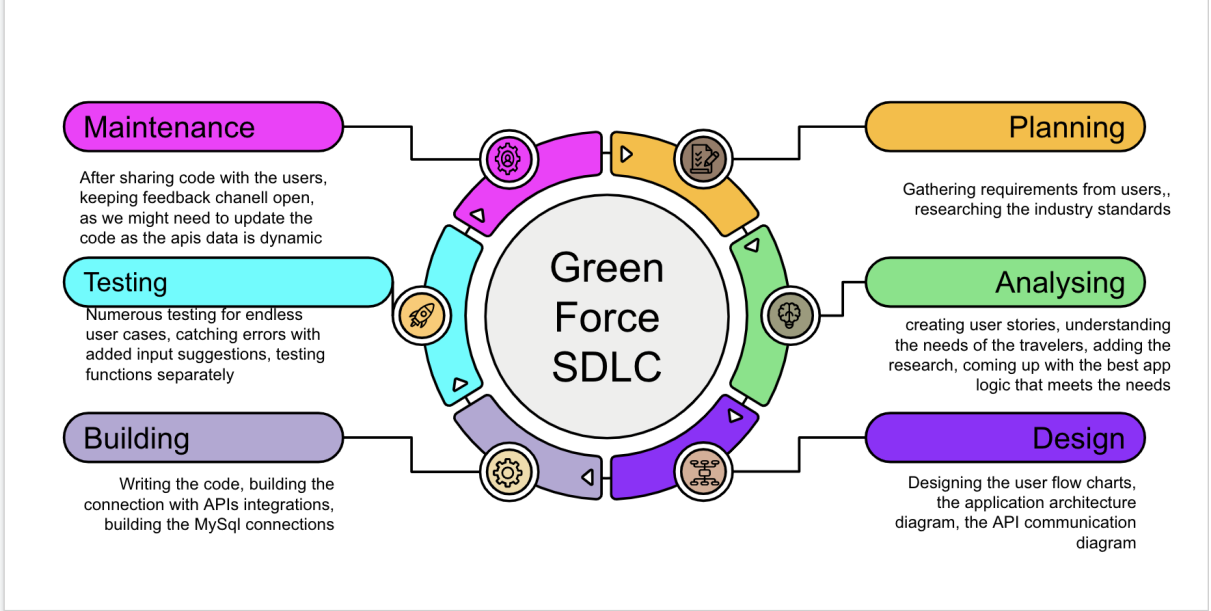
3. Designing architecture - A Design Document Specification (DDS) is created using the specifications outlined in the SRS. Within the DDS, multiple product architecture designs outlined. Market analysts and stakeholders assess the designs in the DDS, and select the most appropriate design to take forward.

4. Developing the product/Building - Developers begin basic work on the project writing the actual code.

5. Product testing and integration - Once the basic product has been developed, more thorough testing of the programs can take place. Small amounts of testing takes place throughout all stages of the software development life cycle, but more thorough testing takes place at this stage. All identified issues are tracked, amended, caught and retested.

6.Maintenance - After thorough testing and amendments, the final product is released in phases. Testing continues in real-world conditions. Feedback is obtained from users and stakeholders as the product is used, and improvements are continually made in line with this feedback.

Documentation, training and support is also an essential part of the process. This is a critical part of the SDLC, as thorough and clear documentation allows for smooth maintenance and improvement to the product.



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

Waterfall methodology is a linear form of project management, it works best for the projects where the end result snd expectations are clearly established at the beggining of the project. The deliverables of each stage are clearly specified and steps need to be acpmplished in the orded.

Agile is in contrary an iterative methodology that incorporates a cyclic and collaborative process.

Criteria	**Agile**	**Waterfall**
approach	Frequent stakeholder interaction	Hands-off; goals and outcome established from the beginning
requrements	Team initiative and short-term deadlines	Completing deliverables to progress to the next phase
flexibility	high	low
Advantages:	-Short-term deadlines encourage productivity and efficiency -The approach is is client-facing, there is a lot of flexibility to change project direction and experiment with new directions based on feedback	-Provides a concrete plan of the project from start to finish -The team establishes project requirements early on, which can save time Workflow is more structured
Disadvantages:	-Because team members are working on multiple phases at a time, there is potential for overlap or unnecessary effort spent on later stages if an early phase needs to be modified	-The implementation process can take longer because of linear approach -You might not realize an issue with a phase until you have already progressed to the next one. This

	-The project timeline is more difficult to determine from the start, and it is also more susceptible to change -Deliverables are not a requirement to progress to the following phase. It can be harder to ensure the entire team is on the same page	would mean going back through each phase and checking where the mistake or error occurred, which can be a timely process. -Rigid, any changes implementation is a complex process
budget	flexible	fixed
client involment	clients is involved in the project development at every step	once the end goal is established, Waterfall does not involve the client or project owner during the process, apart from specific check-ins or for deliverables.

7. **What is a reduced function used for?**

The **reduce()** function executes a technique called reduction. reduce() works by applying a two-argument function to the items of iterable in a loop from left to right, ultimately reducing iterable to a single cumulative value.

How it works in steps:

1. Applies a function to the first two items in an iterable. Gets first partial result.

2. Repeats this step, but next time applies the function to two different items: the partial result from the previous step, and the third item in the iterable. This generates a new partial result.

3. The process is repeated until there are no more iterables, and a single cumulative value results.

To initiate the reduce() function we neet to import it from functools.

Python

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

This function would act as the following regular function:

```
def reduce(function, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = function(value, element)
    return value
```

We can use reduce() with:

1. A user-defined function.
2. A lambda function.
3. A function called operator. mul()

8. **How does merge sort work**

Merge sort is a recursive 'divide and conquer' algorithm. It can be used on an array to divide it into two halves, call itself on each of the two halves, then merge the two sorted halves.

Sequence of actions:

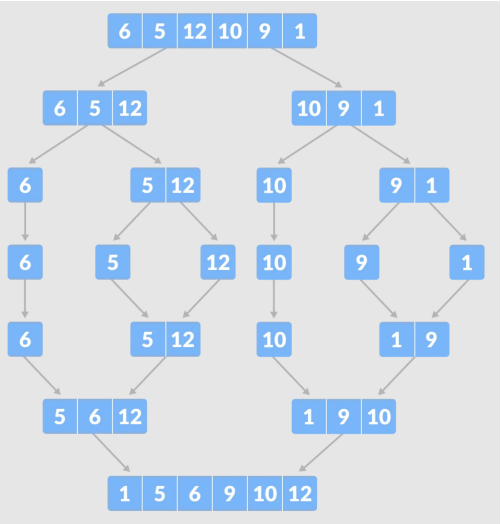
1. Finds the middle point in the array, to use this as a marker for where to split the array in two.

2. Call mergeSort for the first half

3. Call mergeSort for the second half

4. Merge the two halves by calling merge()

The algorithm is recursive, so it will continue to call itself in the way described above until it divided to the stage that each of the two halves of the arrays in each call are size 1. When the array size reaches 1, the merge process begins.



from programiz.com

Merge sort is useful for sorting linked lists, inversion count problem, e-commerse, external sorting

Disadvantages of merge sort:

- Needs additional memory space of 0(n) size to store the new temporary array in
- If array is sorted already, the algorithm will still complete the whole process
- If the task is small, some other algorithms may be faster.

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?**

A generator function is similar to a standard function, but it generates a value and does so using the 'yield' keyword rather than 'return'. If a function's body contains the 'yield' keyword, it is automatically considered a generator function.

The generator function will return an iterable object. One example for the use of generator functions is the generation of Fibonacci numbers. They are sequences of numbers where each number in the sequence equals the sum of the two preceding numbers in the sequence.

```
# A simple Fibonacci sequence generator
def fibonacci(limit):
# Initialize first two Fibonacci Numbers
a, b = 0, 1
# One by one yield next Fibonacci Number
while a < limit:
yield a
a, b = b, a + b
print("\nUsing for in loop")
for i in fib(40):
print(i)
```

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

    In a nutshell, a decorator takes in a function, adds some functionality to this function and then returns it. Functions and methods are called ‘callable; objects as they can be called. A decorator is a callable that returns a callable.

    The decorator can be expressed as a nested function or with the use @ symbol along with the name of the decorator function. The two following notations will provide the same input:

```python
#a)
def cool_decorator(func):
    def inner():
        print("Added colors and functionality")
        func()
    return inner


def targetfunction():
    print("Initial function state")

nicer_version = cool_decorator(targetfunction)
print(nicer_version())
#b)
@cool_decorator
def targetfunction():
    print("Initial function state")

print(cool_decorator(targetfunction))
```

We can both create our own decorators and use the existing ones from the standard libries.

Useful decorators:

@count_calls

This decorator can be used to provide information as to how many times a function is used in software. This function can be useful for debugging. When added to a given function, we will receive an output telling us how many times the function has been ran each time it runs. Can be found in the standard library.

```
from decorators import count_calls
@count_calls
def function_example():
    print("CFG Degree is almost over!")
function_example()
function_example()
function_example()
#output3
```

@dataclass

This decorator can be used to write common standard methods that are typically found in the classes in a speedy manner.

```
from dataclasses import dataclass@dataclass
class CFGStudents:
    name: str
    grade: float
    subject: int = 0

    def total_grade(self) -> float:
        return(self.grade * self.subject)
```

This decorator will automatically create an initialization function, __init__(), with the positional arguments that we need to fill the data in our class. They will also be provided to self automatically, so we only need some data argument to initiate the class.

```
from dataclasses import dataclass

@dataclass
class CFGStudent:
    name: str
    total_points: float
    subject_number: int = 0

def total_grade(self) -> float:
        return (self.total_points / self.subject_number)

alline = CFGStudent("Alline",290,4)
print(total_grade(alline))
#output 72.5
```

@register

The register decorator names a function to be ran at termination. For example, with some software that needs to save whenever we exit.

```
from atexit import register
@register
def final_words():
    print(" May the force be with you!")
```