FINAL ASSESSMENT

Python Software Stream, assessment test 2 hours

- You are given this PDF doc with Assessment tasks description, as well as separate **final_assessment.py** file, which contains sample skeleton code or tests for all parts of this test.
- Rename **final_assessment.py** to **<your-name>_final_assessment.py** and write all your answers/code in this renamed file.
- You will be submitting **<your-name>_final_assessment.py** at the end of 2 hours.
- It is a closed book exam.
- You are allowed to use PyCharm for this assessment.

1. Python OOP and classes	**25 points**

Design a parent class called CFGStudent. It should have general attributes like name, surname, age, email, student id and methods to fetch student's full name and student's id.

Important: there should be an option to pass student id when a new class object is generated, HOWEVER, if no id is passed, then student_id should be automatically generated and assigned to the class.
Design a child class called NanoStudent, which inherits from CFGStudent class. This class should have exactly the same attributes as its parent class, as well as additional two called 'specialization' and 'course grades'.

The child class 'generate_id' method should override its parent method to add the suffix 'NANO' in front of the id.

New methods 'add_new_grade' and 'get_course_grades' should be added. Please see the skeleton structure in the **final_assessment.py** file. You can use it as a skeleton code for your classes OR adjust it and create your own.

SEE ADDITIONAL COMMENTS in the file.

2. Algorithms: find the sum of the even Fibonacci numbers up to the given index.	**25 points**

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

The Fibonacci numbers, commonly denoted ***Fn*** (basically F stands so Fibonacci and **n** means number aka 'index') form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.

TASK:

Given an index limit number, find corresponding Fibonacci values for every index in the limit range (excluding the limit itself) and then sum up all even values in your Fibonacci sequence.

EXAMPLE:

- Given limit = 10
- Corresponding index values within the limit are:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}	F_{20}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765

- Even Fibonacci values within the limit are:

[0, 2, 8, 34]

- Sum of the even numbers and the final result:

0+2+8+34 = **44**

HINTS:

1. Start with writing a stand alone function that can return n'th Fibonacci number
2. List comprehension can be useful here

3. Problem solving coding task - "Validate Subsequence"	**25 points**

Given two non-empty arrays of integers, write a function that determines whether the second array is a subsequence of the first one.

NOTE: A subsequence of an array in this case is a set of numbers that aren't necessarily adjacent in the array, but are in the same order as they appear in the array. Example:

- Array = [1,2,3,4]
- Numbers = [1,3,4] - valid subsequence
- Numbers = [2,4] - valid subsequence
- Numbers = [0,1,2,3,4] - invalid subsequence
- NB: a single number in an array and the array itself are both valid subsequences of the array.

HINTS (only recommendations - feel free to follow your own algorithm):

1. You can solve this question by iterating through the main input array once. 2. Iterate through the main array and look for the first integer in the potential subsequence. If you find that integer, keep on iterating, but now look for the second integer. If you don't find the first integer, is it worth continuing?
3. To actually implement what hint #2 describes you may want to declare a variable holding your position in the potential sequence.

SEE EXAMPLE TESTS in **final_assessment.py** file.

4. Code review challenge: THIS IS A WRITTEN TASK NOT CODING. Imagine that your colleague or a classmate asks you to review a block of code (provided below) for them. **How can this code be improved? Is there anything missing or maybe it needs to be refactored. Please write down your recommendations for this review.**	**25 points**

NOTES:

● We assume that the passed in db_engine object on the class initiation is correct (there is nothing wrong with it). It is an example only.

HINTS:

● Think of SOLID principles and general 'good code keeping' rules.

```python
class Employee:

    def __init__(self, name, is_active, department, id_, db_engine):
        self.name = name
        self.active_status = is_active
        self.department = department
        self.id = id_
        self.db_engine = db_engine

    def update_department(self, department_name):
        self.department = department_name

    def update_status(self, new_is_active):
        self.active_status = new_is_active

    def save_employee(self):
        self.db_engine.save(
            self.name,
            self.active_status,
            self.department,
            self.id
        )

    def remove_employee(self):
        self.db_engine.delete(
            self.name,
            self.id
        )

    def print_employee_report(self, path):
        with open(path, 'w') as file:
            file.writelines(
                [
                    self.name,
                    self.active_status,
                    self.department,
                    self.id
                ]
            )
```

SOLID

S = Single Responsibility Principle

Classes should only change once/be responsible for just one thing. This snippet of code changes the database of Employee class more than once. For example, the final function print_employee_report of the code.

O = Open/Closed Principle

The Employee class in this code should be open for extension, but closed for modification. Therefore, functionality can be added, but the existing code structure should not be changed. This is to avoid bugs and long hours of testing.

L = Liskov Substitution Principle

“Derived classes must be substitutable for their base classes”. For example, when a subclass redefines a function within the parent class, it should not change the behaviour and should act just like a substitute for the parent class.

I = Interface Segregation Principle

This principle highlights that client-specific interfaces are better than general purpose ones. This comes into play with inheritance where subclasses inherit methods from a base class it doesn’t need. For example, the print_employee_report may be running employees that have been removed from the remove_employee function.

D = Dependency Inversion Principle

To avoid any issues when the format of information/data within a function changes, a third interface, like an API should be used to manage this. This could be used in place of the database to create cleaner and clearer output no matter the changes and updates made to the Employee class.