



**FOUNDATION ASSESSMENT II - 2 HOURS**

|  |  |
| --- | --- |
| **SECTION** | **MARK** |
| **Theory Questions** | 31 |
| **Concept Questions** | 19 |
| **Python Challenge** | 25 |
| **SQL Challenge** | 25 |
| 1. **TOTAL** | **100** |

**Important notes:**

* Any code files written must be submitted via a Pull Request to your marker.
* You can submit theory questions through an edited version of this document on Slack, or on the Pull Request by adding python comments into a new file, or using a text or markdown file.
* You are allowed to submit everything on Slack if it is close to the deadline, as long as you work on getting a pull request up soon after.
* It is a closed book exam.
* You are allowed to use PyCharm and MySQL Workbench for this assessment.

**Section 2: Concept Questions [19 marks]**

|  |  |
| --- | --- |
| **2.1 Write a function that takes in an input and checks to see if it’s an**  **isogram. The function should return True or False.**  An isogram is a word where no letter is repeated.  Examples include:   * "isogram" * "uncopyrightable" * “ambidextrously”   #creating function that will return True if the word if isogram\ #and False if the word is not one def isogram(examples):  # need to use lower() so our function is case in-sensitive  examples = examples.lower()  for char in examples:  if examples.count(char) > 1:  return False   return True  examples = "isogram" #True # examples = "madam" #False # examples = "uncopyrightable" #True # examples = "ambidextrously”" #True print(isogram(examples)) | **7 marks** |

|  |  |
| --- | --- |
| **2.2 Make a new test file and write comprehensive unit tests for the**  **function you wrote in 2.1**  For each test case add a comment stating why you chose that case.  import unittest  from isogram\_f import isogram   class TestIsogram(unittest.TestCase):  #testing that our function gives us True statement when we enter isogram  def test\_isogram(self):  self.assertTrue(isogram("isogram"))  self.assertTrue(isogram("uncopyrightable"))  self.assertTrue(isogram("ambidextrously”"))    #testing that we get False when word isn't and isogram  def test\_isogram\_2(self):  self.assertFalse(isogram("madam"))  self.assertFalse(isogram("tintin"))  self.assertFalse(isogram("simmetry"))  if \_\_name\_\_ == '\_\_main\_\_':  unittest.main() | **12 marks** |

**Section 3: Python Challenge [25 marks]**

You are tasked with calculating the minimum classes we need to have so we know how many people to employ. Write a function which when given a number of students, calculates and prints out a string for your proposed number of classes, and a dictionary showing the allocation.

***Key Constraints:***

* The maximum size of a class is 30
* There needs to be a minimum of 2 classes
* The distribution of each class should be as even as possible.
* We want to hire as little people as possible - so where possible focus on bigger classes, and less of them!

***Inputs/Outputs***:

* If 31 was the input, the output would be:

|  |
| --- |
| Proposed Allocation: 2 classes  {'Class 1': 16, 'Class 2': 15} |

* If 59 was the input, the output would be:

|  |
| --- |
| Proposed Allocation: 2 classes  {'Class 1': 30, 'Class 2': 29} |

* If 87 was the input, the output would be:

|  |
| --- |
| Proposed Allocation: 3 classes  {'Class 1': 29, 'Class 2': 29, 'Class 3': 29} |

number\_of\_students = 59  
   
def minimum\_classes(number\_of\_students):  
 # how many classes   
 if number\_of\_students <= 60:  
 number\_of\_classes = 2  
 else:  
 number\_of\_classes = number\_of\_students//30 + (number\_of\_students % 30 > 0)  
 #print(number\_of\_classes)  
  
  
 #allocating students   
 students\_per\_class = number\_of\_students//number\_of\_classes  
 #print(students\_per\_class)  
 # find students left  
 students\_left = number\_of\_students % number\_of\_classes  
 #print(students\_left)  
  
  
 # create dictionary  
 proposed\_allocation= {}  
 #use loop to go through range of classes and distribute students  
 for i in range(number\_of\_classes):  
 class\_size = students\_per\_class  
 #redistribute leftover students  
 if students\_left > 1:  
 class\_size = class\_size + 1  
 students\_left = students\_left - 1  
 #not sure how to format dictionary proposed\_allocation = class\_size doesn't work  
  
 # printing outputs  
 print(f"Proposed Allocation: {number\_of\_classes} classes")  
 print(proposed\_allocation)  
  
  
print(minimum\_classes(number\_of\_students))

**Section 4: SQL Challenge [25 marks]**

In this section you will be fleshing out a database and performing queries.

**Starter Code:**

CREATE DATABASE foundation\_assessment\_ii;

USE foundation\_assessment\_ii;

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **4.1 Write (and execute) syntax to create the following tables:**  Example data is included to help you choose suitable data types  **A] *movie\_info*** *Table*   |  |  |  |  | | --- | --- | --- | --- | | Movie\_ID | Movie\_Name | Movie\_Length | Age\_Rating | | 1 | The Movie | 1:35:00 | 12A |   **B] *screens*** *Table*     |  |  | | --- | --- | | Screen\_ID | Four\_K | | 1 | False |   **C] *showings*** *Table*   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Showing\_ID | Movie\_ID | Screen\_ID | Start\_Time | Available\_Seats | | 1 | 1 | 1 | 12:00:00 | 23 | | **10 marks** |

**Answer4.1:**

**a)**

CREATE TABLE movie\_info(

Movie\_ID integer,

Movie\_Name varchar(50) NOT NULL,

Movie\_Length varchar(50) NOT NULL,

Age\_Rating varchar(50) NOT NULL

);

**b)**

CREATE TABLE screens(

Screen\_ID integer,

Four\_K bool NOT NULL

);

**c)**

CREATE TABLE showings(

Showing\_ID integer,

Movie\_ID integer,

Screen\_ID integer,

Start\_time time,

Available\_Seats integer

);

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **4.2 Write a query to return the name and time of all movies that play after**  **12:00 given there is at least 1 available seat. Display the results in time**  **order.**  SELECT movie\_info.movie\_name, showings.start\_time  FROM movie\_info  JOIN showings ON movie\_info.movie\_ID = showings.movie\_ID  WHERE showings.start\_time >= '12:00:00' AND showings.available\_seats > 0  ORDER BY showings.start\_time;  Output:   |  |  | | --- | --- | | La Allure | 20:00:00 | | La Allure | 23:00:00 | | The 3D Amazing Movie | 14:00:00 | | The 3D Amazing Movie | 17:00:00 | | The Cartoon | 12:30:00 | | The Coming Of Age | 12:00:00 | | The Movie | 12:00:00 | | The Movie | 17:00:00 | | The Scary Cartoon | 13:30:00 | | The Scary Cartoon | 17:00:00 | | The War | 15:00:00 | | **6 marks** |

|  |  |
| --- | --- |
| **4.3 Return the name of the movie with the most showings.**  1) first I’ve created table to count the movie showings  CREATE TABLE movie\_showings\_count AS  SELECT movie\_ID, COUNT(\*) AS num\_showings  FROM showings  GROUP BY movie\_ID;  2) then selected the movie with the most showings by joining movie\_info with movie\_showings\_count and usiting limit to select 1 at the top  SELECT movie\_info.Movie\_Name  FROM movie\_info  INNER JOIN movie\_showings\_count  ON movie\_info.Movie\_ID = movie\_showings\_count.movie\_ID  ORDER BY movie\_showings\_count.num\_showings DESC  LIMIT 1;  **Answer: 'The 3D Amazing Movie'** | **9 marks** |