Python Programming for Petroleum Engineers

HW1 (Pyhton Hands-on)

Welcome to this comprehensive set of Python exercises tailored specifically for petroleum engineering students! These exercises are designed to challenge your understanding of Python programming concepts and their applications in the oil and gas industry.

Throughout this notebook, you will encounter a wide range of exercises that cover various Python topics, including data structures, control flow statements, functions, modules, NumPy arrays, Pandas dataframes, and data visualization with Matplotlib. Each exercise is carefully crafted to incorporate real-world petroleum engineering scenarios, ensuring that you gain practical experience in applying Python to solve problems relevant to your field.

Whether you're a beginner or an experienced programmer, these exercises will help you strengthen your Python skills and develop a deeper understanding of how programming can be leveraged to tackle complex problems in the petroleum industry.

In addition to coding exercises, you will also encounter debugging challenges and error handling scenarios, which will help you develop the critical thinking skills necessary for effective troubleshooting and problem-solving.

Remember, programming is a continuous learning process, and these exercises are meant to be challenging. Don't hesitate to seek help from your instructors, classmates, or online resources when needed. Collaboration and knowledge sharing are key elements of successful programming. But first try to challenge your brain improve your skills.

So, let's embark on this exciting journey and dive into the world of Python programming for petroleum engineers!

Task 1: Well Distances Calculation

In this task, you will work with well coordinates and calculate the distances between pairs of wells.

Part 1: Generate Well Coordinates

1. Use the provided function `generate_well_coordinates(student_id)` with your student ID as input to generate 10 random (x, y) tuples representing well coordinates.
2. Print the list of well coordinates to ensure you have the correct data.

Part 2: Calculate Well Distances

1. Create a nested list (10 x 10) to store the distances between each pair of wells.
2. Use the Pythagorean theorem to calculate the distance between each pair of wells, and store the distances in the nested list.
3. Print the nested list of well distances.

Part 3: Find Minimum and Maximum Distances

1. Write a function to take well coordinates list as input then find and returns the two wells with the minimum distance between them, and print their indices and corresponding distance.
2. Write a function to take well coordinates list as input then find and returns the maximum distance between them, and print their indices and corresponding distance.

Extra Points: Well Coordinates Plot Plot the well coordinates and draw the two wells with least distance with different colors on it.

In [ ]:
```python
import random

def generate_well_coordinates(student_id):
    """
    Generates 10 random (x, y) tuples representing well coordinates.

    Args:
        student_id (int): The student ID to be used as the seed for the random numb

    Returns:
        list: A list of 10 (x, y) tuples representing well coordinates.
    """

    # Set the seed for the random number generator
    random.seed(student_id)

    # Generate 10 random (x, y) tuples
    well_coordinates = [(random.randint(-1000, 1000), random.randint(-1000, 1000))

    return well_coordinates

# Example usage
student_id = 12345678  # Replace with your student ID
well_coords = generate_well_coordinates(student_id)
print(well_coords)
```

In [ ]:
```python
# Write your code here:
```

Task 2: Well Distance Classification

In this task, you will create a function to classify the distances of wells from an arbitrary point based on predefined distance ranges.

Part 1: Define the Classification Function

1. Define a function `classify_well_distances(well_coords, point)` that takes two arguments:
    - `well_coords` : A list of (x, y) tuples representing the coordinates of wells.
    - `point` : A tuple (x, y) representing the arbitrary point from which distances will be calculated.
2. Inside the function, create an empty list called `well_classifications` .
3. Iterate through the `well_coords` list.
4. For each well coordinate (x, y), calculate the distance from the `point` using the Pythagorean theorem.
5. Based on the calculated distance, append a string classification to the `well_classifications` list using the following criteria:
    - If the distance is less than or equal to 500 meters, append "Nearby"
    - If the distance is greater than 500 meters but less than or equal to 1000 meters, append "Moderate"
    - If the distance is greater than 1000 meters, append "Far"
6. Return the `well_classifications` list from the function.

Part 2: Test the Function

1. Use the `generate_well_coordinates(student_id)` function with your student ID to generate a list of 10 well coordinates.
2. Define an arbitrary point, for example, `point = (100, 200)` .
3. Call the `classify_well_distances` function with the generated well coordinates and the arbitrary point, and store the result in a variable.
4. Print the resulting list of well classifications.

This task challenges students to define a function that performs distance calculations and applies conditional statements to classify the distances into predefined categories. It also reinforces the concepts of iterating through lists, using the Pythagorean theorem, and working with tuples and lists.

By testing the function with their own well coordinates and an arbitrary point, students can verify the correctness of their implementation and gain practical experience in applying these concepts to petroleum engineering scenarios.

In [ ]:
```
# Write your code here:
```

Task 3: Reservoir Pressure Analysis

You are working for an oil and gas company that operates multiple reservoirs. Each reservoir is divided into grid blocks, and you have access to the pressure data for each grid block. Your task is to analyze the pressure data and provide insights into the reservoir performance.

The pressure data is stored in a NumPy array, where each row represents a grid block, and each column corresponds to a specific time step. The array has the following shape: `(num_grid_blocks, num_time_steps)`, where `num_grid_blocks` is the number of grid blocks in the reservoir, and `num_time_steps` is the number of time steps for which pressure data is available.

Your program should perform the following tasks:

1. Calculate the pressure gradient between adjacent grid blocks at each time step. The pressure gradient is the difference in pressure between two neighboring grid blocks. Store the pressure gradients in a new NumPy array with the same shape as the input pressure data array.
2. Identify the grid blocks with the highest and lowest pressure gradients at the last time step, and print their indices and corresponding pressure gradients.
3. Calculate the average pressure gradient across the entire reservoir at each time step, and store the results in a new NumPy array with shape `(num_time_steps,)`.
4. Plot the average pressure gradient over time using Matplotlib. Add a title, x-label, and y-label to the plot.
5. Implement a function `detect_anomalies(pressure_gradients, threshold)` that takes the pressure gradient array and a threshold value as input. The function should return a boolean mask indicating the grid blocks where the pressure gradient exceeds the threshold at any time step.
6. Apply the boolean mask from step 5 to the original pressure data array to extract the anomalous grid blocks, and print the indices of these grid blocks.

Use the provided function `generate_reservoir_data(student_id)` with your student ID as input to generate the pressure data array.

In [ ]:
```
import numpy as np

def generate_reservoir_data(student_id):
    """
    Generates pressure data for a reservoir with a fixed number of grid blocks and

    Args:
        student_id (int): A 9-digit integer representing the student's ID.

    Returns:
        numpy.ndarray: A 2D NumPy array representing the pressure data for the rese
                       The array has shape (num_grid_blocks, num_time_steps).
```

```
    """
    # Set the random seed based on the student ID
    np.random.seed(student_id)

    # Define the number of grid blocks and time steps
    num_grid_blocks = 50
    num_time_steps = 100

    # Generate random pressure data for the reservoir
    pressure_data = np.random.randint(100, 500, size=(num_grid_blocks, num_time_ste

    return pressure_data
```

In [ ]:
```
# Write your code here:
```

Task 4: Reservoir Grid Block Analysis

Description: In petroleum reservoir management, reservoirs are divided into grid blocks to model fluid flow and pressure distribution. You are tasked with analyzing pressure and porosity data across a 3D reservoir grid to identify critical regions for drilling. This task focuses on multi-dimensional array manipulation, nested loops, and statistical computations using NumPy.

Parts:

1. **Data Generation**:
    - Use the provided function `generate_reservoir_grid(student_id)` to generate two 3D NumPy arrays:
        - `pressure_grid`: Shape `(10, 10, 5)` representing pressure (in psi) across a 10x10x5 grid (x, y, z dimensions).
        - `porosity_grid`: Shape `(10, 10, 5)` representing porosity (in percentage) across the same grid.
    - Compute and print the mean, standard deviation, minimum, and maximum pressure and porosity across the entire grid.
    - Compute and print the mean, standard deviation, minimum, and maximum pressure and porosity across each layer (z dimension).
2. **High-Pressure Zones**:
    - Identify grid blocks where the pressure exceeds 400 psi. Create a boolean array of the same shape as `pressure_grid` with `True` for blocks meeting this condition and `False` otherwise.
    - Count the number of high-pressure blocks and print their coordinates (i, j, k) and corresponding pressure values.
3. **Porosity-Pressure Correlation**:
    - For each z-layer (k=0 to 4), compute the element-wise product of pressure and porosity values in that layer (shape: `(10, 10)`). Store these products in a new 3D

array of shape `(10, 10, 5)`.

- Calculate the average product per layer and print these values. Identify the layer with the highest average product and print its index.

4. **Visualization**:

   - For the z=0 layer, create a heatmap using Matplotlib to visualize the pressure values. Use a colorbar and label the axes as "X Coordinate" and "Y Coordinate". Add a title "Pressure Distribution at z=0".

Provided Code:

```
import numpy as np

def generate_reservoir_grid(student_id):
    """
    Generates 3D pressure and porosity grids for a reservoir.

    Args:
        student_id (int): Student ID for random seed.

    Returns:
        tuple: (pressure_grid, porosity_grid) where each is a (10, 10, 5)
array.
    """
    np.random.seed(student_id)
    pressure_grid = np.random.uniform(200, 500, size=(10, 10, 5))
    porosity_grid = np.random.uniform(5, 25, size=(10, 10, 5))
    return pressure_grid, porosity_grid
```

Task 5: Working with CSV data

For this task you should import well-logging data from a CSV file using pandas and identify the pay zone. You are required to identify the pay zone by applying a set of cutoffs and calculate the height of the pay zone. Finally, you should present your findings in a clear and concise manner.

Along with the Homework notebook, there is a CSV data file (`wellDATA.csv`). The file contains well-logging data, which includes four columns: `DEPTH`, `PHIE`, `SWE`, and `VSH`. These columns represent the depth of the well, porosity, effective water saturation, and volume of shale respectively.

First you should import this dataset using PANDAS library and getting in touch with it by examining it.

```
import pandas as pd

df = pd.read_csv('wellDATA.csv') # This will load the data into pandas
dataframe object
df.head() # This will print some first rows of data for examination
```

Then you should identify each depth as payzones or non-payzones by applying the following cutoffs:

- PHIE >= 7%
- SWE <= 30%
- VSH <= 30%

The result should be stored in a new column of your dataframe (`df`) assigning `1s` for payzones and `0s` for non-payzones.

Next you should calculate the height of the payzones through the result of the last step and report it.

Hints:

- **Do not forget Google to learn concepts you do not know**
- You can convert each colomn of pandas dataframe into numpy array by the following code:

  ```
  PHIE = df['PHIE'].to_numpy()
  ```
- You can add new columns to your dataframe like this:

  ```
  #Assume "payzone_flag" is the numpy array containing the payzones and
  non-payzones you have identified.
      df['payzone_flag'] = payzone_flag
  ```

In [ ]:
```
# Write your code here:
```

Task 6: Plotting

1. Import the `wellDATA.csv` using `PANDAS` library. Plot **scatters** of PHIE vs. DEPTH, SWE vs. DEPTH, VSH vs. DEPTH and, payzone_flag vs. DEPTH using `matplotlib` . Note: All of the plots should be placed in `one figure` and each log should be displayed in `a column` (so you figure should have `4` columns).
2. Plot 3D scatter plot of (SWE, VSH, DEPTH) color the points with payzone_flag value and size the points with the value of PHIE

Your plots should be titled and their axis should be labeled appropriately.

In [ ]:
```
# Write your code here:
```

Task 7: Debugging

Each of the following code cells has one or more erros `Find them` , `fix them` and `write the reason` causes the error.

In [ ]:
```python
import numpy as np

def matrix_operations(A, B, C, D, E):
    # Step 1: Matrix multiplication of A and B
    result1 = A @ B

    # Step 2: Elementwise multiplication of result1 and C
    result2 = result1 * C[:, :result1.shape[1]]

    # Step 3: Matrix multiplication of result2 and D
    result3 = result2 @ D[:-1]

    # Step 4: Elementwise multiplication of result3 and E
    result4 = result3 * E.T

    # Step 5: Calculate the sum of result4
    final_result = np.sum(result4, axis=1)

    return final_result

# Test case
A = np.random.randint(1, 10, size=(5, 3))
B = np.random.randint(1, 10, size=(3, 4))
C = np.random.randint(1, 10, size=(5, 5))
D = np.random.randint(1, 10, size=(4, 2))
E = np.random.randint(1, 10, size=(2, 3))

result = matrix_operations(A, B, C, D, E)
print("Final result:", result)

# Your tasks:

# 1. Identify and fix the bugs in the provided code to ensure that the matrix opera
# 2. Explain the issues that caused the bugs.
# 3. Provide the correct output for the given test case after fixing the bugs.
```

In [ ]:
```python
''' The following program is supposed to take a list of numbers as input and return
    that contains only the odd numbers in the original list.
    However, there is a bug in the program that causes it to return an empty list.
    identify and fix the bug?'''

def filter_odd_numbers(numbers):

    for num in numbers:
        odd_numbers = []
        if num % 2 != 0:
            odd_numbers.append(num)
    return odd_numbers

# Test the function
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = filter_odd_numbers(numbers)
print(result)
```

In [ ]:

```
'''
The following code should perform these tasks:

    1- Calculates the sum of all the elements in the array.
    2- Calculates the product of all the even elements in the array.
    3- Replaces all odd elements in the array with -1.

But there are some bugs in it
'''

import numpy as np

# Create a numpy array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Initialize sum and product variables
total_sum = 0
even_product = 0

# Iterate over the array using a for loop
for ind in range(1,len(arr)):

    element = arr[ind]


    total_sum += element

    if element % 2 == 0:
        even_product *= element

    if element % 2 != 0:
        selected = (arr == element)

    arr[ind] = -1


# Print the results
print("Sum of all elements:", total_sum)
print("Product of all even elements:", even_product)
print("Array after replacing odd elements with -1:")
print(arr)
```

In [ ]:

```
"""
This code should calculate the density of each oil sample and print each of them in
Its result should be:
    The density of the oil sample is: 1000.0000000000001 g/cm^3
    The density of the oil sample is: 894.1176470588235 g/cm^3
    The density of the oil sample is: 1100.0 g/cm^3
"""



oil_samples = [[820, 0.82], [760, 0.85], [880, 0.80]] # This array contains the mas

def calculate_density(mass, volume):
    density = mass / volume
```

```
    return

for sample in oil_samples:
    mass, volume = oil_samples[0]
density = calculate_density(mass, volume)
print("The density of the oil sample is:", density, "g/cm^3")
```

In [ ]:
```
# The function minimum() should find the minimum of a given list but it doesn't see


def minimum(input_list):
    a = 0
    for x in range(1, len(input_list)):
        if input_list[x] < a:
            a = input_list[x]
    return a

a = [3, 4, 2, 6, 1, 7, 7, 8]

print(minimum(a))
```

In [ ]:
```
'''
The following code tries to remove all elements of the list wich starts with the le
but the 'three' have remained in the printed results why? '''

nums = ['zero', 'one', 'two', 'three', 'four']

for num in nums:
    if num.startswith('t'):
        nums.remove(num)

print(nums)
```

In [ ]:
```
import numpy as np

def calculate_reservoir_volume(thickness, area, porosity):
    """
    Calculates the total pore volume of a reservoir grid.
    Args:
        thickness: 2D array of shape (n, m) with grid block thicknesses (ft).
        area: 2D array of shape (n, m) with grid block areas (sq ft).
        porosity: 2D array of shape (n, m) with porosity percentages.
    Returns:
        Total pore volume (cu ft).
    """
    volume = 0
    for i in range(thickness.shape[0]):
        for j in range(thickness.shape[1]):
            volume += thickness[i, j] * area[i, j] * porosity[i]
    return volume

# Test case
thickness = np.ones((5, 5)) * 10
area = np.ones((5, 5)) * 1000
porosity = np.full((5, 5), 0.2)
```

```
volume = calculate_reservoir_volume(thickness, area, porosity)
print("Total pore volume:", volume, "cu ft")



'''
- Fix the code to compute the total pore volume correctly (Pore volume is calculate
 per grid block, summed over all blocks).
- Explain each error and its impact.
- Provide the corrected output for the test case (should be 10000 cu ft).
'''
```

In [ ]:

```
import numpy as np

def smooth_production_rates(rates):
    """
    Applies a 3-day moving average to smooth production rates.
    Args:
        rates: 1D array of daily production rates.
    Returns:
        Smoothed rates array.
    """
    smoothed = np.zeros_like(rates)
    for i in range(len(rates)):
        start = max(0, i - 1)
        end = min(len(rates), i + 1)
        smoothed[i] = np.mean(rates[start:end])
    return smoothed

# Test case
rates = np.array([100, 120, 110, 130, 125, 90, 140, 130, 110, 80])
smoothed = smooth_production_rates(rates)
print("Smoothed rates:", smoothed)



'''
- Fix the code to compute a 3-day moving average correctly.
- Explain each error and its impact.
- Provide the corrected output for the test case.
'''
```

After completing all the tasks, please, `convert it to PDF`, zip it with your answers notebook (`ipynb file`) and send it.

MAKE SURE TO HAVE THE RESULTS OF THE CODES IN YOUR FILES

Have notes and explanations along with your results