practice_midterm_answers

October 11, 2023

[ ]:
```python
### Question 1

def count_words(file_name):
    words = 0
    with open(file_name, 'r') as file:
            for line in file:
                line_words = line.strip().split()
                words = words + len(line_words)
    return words

def organize_books(books: list) -> list:
    # Sort the books based on the number of words in each book (in descending␣
 ↪order)
    sorted_books = sorted(books, key=lambda book: count_words(book),␣
 ↪reverse=True)
    return sorted_books

# Example usage:
books = ["book1.txt", "book2.txt", "book3.txt"]
organized_books = organize_books(books)
print(organized_books)
```

The answer consists of two functions, `count_words` and `organize_books`. These functions work together to organize a list of book filenames based on the number of words in each book in descending order. Here's a step-by-step explanation of the code:

`count_words` Function: This function takes a filename as input and returns the number of words in the file.

1. `def count_words(file_name):`: Defines a function called `count_words` that takes a single argument, `file_name`.
2. `words = 0`: Initializes a variable `words` to store the word count, starting at 0.
3. `with open(file_name, 'r') as file:`: Opens the file specified by `file_name` in read ('r') mode using a `with` statement. This ensures that the file is properly closed after reading, even if an exception occurs.
4. `for line in file:`: Iterates through each line in the opened file.

5. `line_words = line.strip().split()`: For each line, it removes leading and trailing whitespace using `strip()` and then splits the line into words using `split()`. The result is stored in the `line_words` list.
6. `words = words + len(line_words)`: Adds the number of words in the current line to the `words` variable.
7. `return len(words)`: Returns the total word count in the given file as the output of the function.

`organize_books` Function: This function takes a list of book filenames and returns a sorted list of filenames based on the number of words in each book.

1. `def organize_books(books: list) -> list::` Defines a function called `organize_books` that takes a list of filenames (`books`) as input and returns a list of filenames as output. The type hints indicate that `books` is expected to be a list, and the function returns a list.
2. `sorted_books = sorted(books, key=lambda book: count_words(book), reverse=True)`: Sorts the `books` list based on the number of words in each book. The `sorted` function is used with the `key` parameter, which specifies a function (`lambda book: count_words(book)`) to determine the sorting key. The `reverse=True` argument sorts the books in descending order (from most words to fewest words).
3. `return sorted_books`: Returns the sorted list of book filenames as the output of the function.

[ ]:
```python
# 2. (10 points) Write a Python function that takes a string as input and␣
 ↪returns the longest substring without
# repeating characters. For example, if the input string is 'abcabcbb', the␣
 ↪function should return 'abc' because
# it is the longest substring without repeating characters.

def longest_substring(s):
    # Initialize an empty list to store the characters of the current substring
    l = []

    # Initialize variables to keep track of the maximum length and the␣
 ↪corresponding substring
    max_length = 0
    max_substring = []

    # Iterate through the characters in the input string 's'
    for i in range(len(s)):
        # While the current character is not in the 'l' list (no repeating␣
 ↪characters)
        while (i < len(s)) and (s[i] not in l):
            # Add the current character to the 'l' list
            l.append(s[i])
            i += 1
```

```
        # Check if the length of the current substring ('l') is greater than␣
 ↪the maximum length found so far
        if len(l) > max_length:
            # Update the maximum length
            max_length = len(l)
            # Update the maximum substring
            max_substring = l

        # Reset the 'l' list for the next iteration (starting a new substring)
        l = []

    # Return the length of the maximum substring found
    return ''.join(max_substring)
```

[ ]:
```
### Question 3

import random
import string

def generate_pass() -> str:
    # Define the character sets
    special_characters = "!@#$%^&*"
    lowercase_letters = string.ascii_lowercase
    uppercase_letters = string.ascii_uppercase
    digits = string.digits

    # Ensure at least one character from each set
    password = random.choice(special_characters) + random.
 ↪choice(lowercase_letters) + \
               random.choice(uppercase_letters) + random.choice(digits)

    # Generate the remaining characters randomly
    all_characters = special_characters + lowercase_letters + uppercase_letters␣
 ↪+ digits
    password_length = random.randint(8, 16)  # Random length between 4 and 16␣
 ↪(total length will be 8-20)
    while len(password) < password_length:
        password += random.choice(all_characters)

    return password

# Example usage:
password = generate_pass()
print(password)
```

The provided code defines a function `generate_pass()` to generate a random password with specific criteria, and it also includes an example usage to generate and print a random password. Here's a

step-by-step explanation of the code:

`generate_pass` Function: This function generates a random password based on certain criteria:

1. `import random` and `import string`: Imports the `random` module for generating random characters and the `string` module for accessing character sets.
2. `def generate_pass() -> str:`: Defines a function named `generate_pass` that takes no arguments and returns a string (the generated password).
3. Character Sets:
   - `special_characters`: Contains a set of special characters "!@#$%^&*".
   - `lowercase_letters`: Contains all lowercase letters from the alphabet.
   - `uppercase_letters`: Contains all uppercase letters from the alphabet.
   - `digits`: Contains all digits (0-9).
4. Ensure at least one character from each set:
   - The code selects one random character from each of the character sets (`special_characters`, `lowercase_letters`, `uppercase_letters`, and `digits`) and concatenates them together to ensure that the generated password contains at least one character from each of these sets.
5. Generate the remaining characters randomly:
   - The `all_characters` variable is created by combining all character sets.
   - `password_length` is assigned a random integer between 8 and 16 (inclusive). This determines the total length of the password, which will be between 8 and 20 characters.
   - A `while` loop is used to generate additional random characters and append them to the `password` until it reaches the desired length (`password_length`).
6. `return password`: Returns the generated password as a string.

This code ensures that the generated password has a mix of special characters, lowercase letters, uppercase letters, and digits, making it more secure and suitable for various applications.

[ ]:
```python
# 4. (10 points) Given a list of integers nums and an integer target, write a␣
 ↪Python function to find a
# contiguous subarray (a subset of the list) that, when summed, equals the␣
 ↪target. Return the starting and
# ending indices of the subarray. For example, if nums = [2, 7, 11, 15] and␣
 ↪target = 9, the function should
# return [0, 1] because nums[0] + nums[1] = 2 + 7 = 9.

def sum_target(nums, target):
    # Iterate through the elements of 'nums' using 'end' as the end pointer
    for end in range(len(nums)):
        sum = nums[end]  # Initialize the 'sum' with the current element at␣
 ↪'end'
        l = end + 1  # Initialize 'l' as the next element after 'end'
```

```
        # Continue adding elements to 'sum' from 'l' until reaching the end of 'nums'
        while l < len(nums):
            sum += nums[l]

            # Check if 'sum' is less than the 'target'
            if sum < target:
                l += 1
            # Check if 'sum' is equal to the 'target'
            elif sum == target:
                return [end, l]  # Return the indices of the two elements whose sum equals the 'target'
            else:
                break  # If 'sum' exceeds the 'target', break out of the loop and move to the next 'end' element

print(sum_target([1, 4, 0, 0, 3, 10, 5], 7))
```

[ ]:
```
### Question 5

## Ten biggest classes at rutgers
def find_top_ten(classes: dict) -> list:
    # Sort the dictionary by the number of students in each class in descending order
    sorted_classes = sorted(classes.items(), key=lambda x: x[1], reverse=True)
    biggest_classes = list(map(lambda x: x[0], sorted_classes))

    # Extract the top 10 largest classes from the sorted list
    top_ten = biggest_classes[:10]

    return top_ten

# Example usage:
classes = {
    "CS210": 540,
    "EE279": 250,
    "PHIL101": 440,
    # Add more classes here
}

top_ten_classes = find_top_ten(classes)
print(top_ten_classes)

def find_lowest_enrollment_cs_classes(classes: dict) -> list:
    # Filter CS classes with level >= 300 and convert them to a list of tuples (class_name, enrollment)
```

```
    cs_classes = [(class_name, enrollment) for class_name, enrollment in␣
 ↪classes.items() if int(class_name[-3:]) >= 300 and class_name.
 ↪startswith("CS")]

    # Sort the CS classes by enrollment in ascending order
    sorted_cs_classes = sorted(cs_classes, key=lambda x: x[1])
    smallest_classes = list(map(lambda x: x[0], sorted_cs_classes))

    # Extract the ten CS classes with the lowest enrollment
    lowest_enrollment_classes = smallest_classes[:10]

    return lowest_enrollment_classes

# Example usage:
cs_classes = {
    "CS210": 540,
    "CS350": 150,
    "CS400": 80,
    "CS450": 60,
    "CS301": 200,
    "PHIL938": 586
    # Add more classes here
}

lowest_enrollment_cs_classes = find_lowest_enrollment_cs_classes(cs_classes)
print(lowest_enrollment_cs_classes)
```

`find_top_ten` Function: This function finds the top ten classes based on the number of students in each class.

1. `def find_top_ten(classes: dict) -> list:`: Defines a function named `find_top_ten` that takes a dictionary (`classes`) as input and returns a list of the top ten class names.
2. Sorting the dictionary:
   - `sorted_classes = sorted(classes.items(), key=lambda x: x[1], reverse=True)`: Sorts the input dictionary `classes` by the number of students in each class (`x[1]`) in descending order. It returns a list of tuples where each tuple contains the class name and the number of students.
3. Extracting class names:
   - `biggest_classes = list(map(lambda x: x[0], sorted_classes))`: Extracts the class names from the sorted list of tuples and stores them in the `biggest_classes` list.
4. Extracting the top ten:
   - `top_ten = biggest_classes[:10]`: Selects the first ten class names from the `biggest_classes` list to find the top ten classes with the highest enrollment.

5. `return top_ten`: Returns the top ten class names as a list.

`find_lowest_enrollment_cs_classes` Function: This function finds the ten upper level computer science (CS) classes with the lowest enrollment, filtering based on class name criteria.

1. `def find_lowest_enrollment_cs_classes(classes: dict) -> list:`: Defines a function named `find_lowest_enrollment_cs_classes` that takes a dictionary (`classes`) as input and returns a list of the ten CS class names with the lowest enrollment.
2. Filtering CS classes:
   - `cs_classes = [(class_name, enrollment) for class_name, enrollment in classes.items() if int(class_name[-3:]) >= 300 and class_name.startswith("CS")]`: Filters CS classes by checking if their level (last three characters of the class name) is greater than or equal to 300 and if the class name starts with "CS". It converts the filtered classes into a list of tuples with class name and enrollment.
3. Sorting CS classes:
   - `sorted_cs_classes = sorted(cs_classes, key=lambda x: x[1])`: Sorts the CS classes by enrollment in ascending order, using the number of students (`x[1]`) as the sorting key.
4. Extracting class names:
   - `smallest_classes = list(map(lambda x: x[0], sorted_cs_classes))`: Extracts the class names from the sorted list of tuples and stores them in the `smallest_classes` list.
5. Extracting the ten smallest:
   - `lowest_enrollment_classes = smallest_classes[:10]`: Selects the first ten class names from the `smallest_classes` list to find the ten CS classes with the lowest enrollment.
6. `return lowest_enrollment_classes`: Returns the ten CS class names with the lowest enrollment as a list.

[ ]:
```
# 6. (10 points)You have a CSV file named employeeData.csv with the following␣
 ↪columns:
# EmployeeID, FirstName, LastName, Department, Salary, and HireDate.
# Each row represents an employee's information in a company.

# (a) Write Python code to read the employeeData.csv file into a Pandas␣
 ↪DataFrame named "employee_df."
# (b) Calculate and print the following statistics for the Salary column: \
# Mean (average) salary, Median salary and Standard deviation of salaries.
# (c) Find and print the following information:
# - The employee with the highest salary, including their full name and salary.
# - The department with the most employees.
```

```
# (d) Create a new column named YearsWorked in the DataFrame, which represents the number of years each
# employee has been with the company. Assume the current year is 2023, and calculate the YearsWorked based on
# the HireDate column. Then, print the updated DataFrame with the new column included.

# Helper code to create the CSV file

import csv

# Data to be written to the CSV file
data = [
    ["EmployeeID", "FirstName", "LastName", "Department", "Salary", "HireDate"],
    [1, "John", "Doe", "HR", 50000, "2021-01-15"],
    [2, "Jane", "Smith", "Marketing", 60000, "2020-09-10"],
    [3, "Alice", "Johnson", "Engineering", 75000, "2021-03-20"],
    [4, "Bob", "Brown", "Finance", 70000, "2019-12-05"],
    [5, "Mark", "Fitz", "Finance", 80000, "2020-12-05"]
]

# Specify the file name
csv_file_name = "employeeData.csv"

# Write the data to the CSV file
with open(csv_file_name, mode="w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(data)

print(f"The CSV file '{csv_file_name}' has been created with the specified data.")
```

[ ]:
```
#6a)

import pandas as pd
import datetime

# Read the data from a CSV file named "employeeData.csv" and create a DataFrame
employee_df = pd.read_csv("employeeData.csv")

# Print the contents of the DataFrame to the console
print(employee_df)
```

[ ]:
```
#6b)

# Calculate the mean (average) salary of employees from the 'Salary' column in the DataFrame
```

```
mean = employee_df['Salary'].mean()

# Print the calculated mean value
print('Mean Salary:', mean)

# Calculate the median salary of employees from the 'Salary' column in the␣
 ↪DataFrame
median = employee_df['Salary'].median()

# Print the calculated median value
print('Median Salary:', median)

# Calculate the standard deviation of salaries in the 'Salary' column of the␣
 ↪DataFrame
std = employee_df['Salary'].std()

# Print the calculated standard deviation value
print('Standard Deviation of Salary:', std)
```

[ ]:
```
#6c)

# Find the employee with the highest salary:
# - 'employee_df['Salary'].idxmax()' returns the index of the row with the␣
 ↪highest salary.
# - 'employee_df.loc[...]' is used to locate and retrieve the entire row␣
 ↪(employee) with the highest salary.
highest_salary_employee = employee_df.loc[employee_df['Salary'].idxmax()]

# Get the first name and last name of the highest-paid employee from the␣
 ↪retrieved row
employee_name = highest_salary_employee['FirstName'] + ' ' +␣
 ↪highest_salary_employee['LastName']

# Print the full name of the employee with the highest salary
print("Employee with the Highest Salary:", employee_name)


# Find the department with the most employees:
# - 'employee_df['Department'].mode()' returns the mode (most frequently␣
 ↪occurring value) of the 'Department' column.
# - '[0]' is used to access the first (and possibly only) mode value.
department_with_most_employees = employee_df['Department'].mode()[0]

# Print the department with the most employees
print("Department with the Most Employees:", department_with_most_employees)
```

[ ]:
```python
#6d)

current_year = 2023

# Convert the 'HireDate' column to datetime format using 'pd.to_datetime()'
employee_df['HireDate'] = pd.to_datetime(employee_df['HireDate'])

# Calculate the 'YearsWorked' for each employee:
# - 'employee_df['HireDate'].dt.year' extracts the year from the 'HireDate' column.
# - Subtracting the extracted year from the current year calculates the number of years worked.
employee_df['YearsWorked'] = current_year - employee_df['HireDate'].dt.year

# Print the DataFrame with the added 'YearsWorked' column
print(employee_df)
```

[ ]:
```python
### Question 7
import pandas as pd

# Create Employee Sales DataFrame
employee_sales_data = {
    "Employee ID": [139, 170],
    "Sales": [1367, 10967],
    "Costs": [198, 507]
}

employee_sales = pd.DataFrame(employee_sales_data)

# Create Employee Info DataFrame
employee_info_data = {
    "Employee ID": [139, 170],
    "Name": ["Bobby Jindhal", "Noah Baumbach"],
    "Department Name": ["Fishing", "Skiing"],
    "Email": ["b.jindhal@rocketmail.com", "nbthegoat@gmail.com"],
    "Phone": ["898-078-9475", "947-958-0973"]
}

employee_info = pd.DataFrame(employee_info_data)

# 1. Join the two DataFrames together
merged_df = pd.merge(employee_sales, employee_info, on="Employee ID")

# 2. Create a new column for area code
def extract_area_code(phone):
    if pd.notna(phone):
        return int(phone.split("-")[0])
```

```
    else:
        return None

merged_df["Area Code"] = merged_df["Phone"].apply(extract_area_code)

# 3. Calculate department performance and find the winning department
merged_df["Performance"] = merged_df["Sales"] - merged_df["Costs"]
department_performance = merged_df.groupby("Department Name")["Performance"].
 ↪sum()
winner = department_performance.idxmax()

# Print the resulting DataFrame and the winning department
print(merged_df)
print("\nThe winning department is:", winner)
```

Here's a line-by-line breakdown of the answer:

```
# 1. Join the two DataFrames together
merged_df = pd.merge(employee_sales, employee_info, on="Employee ID")
```

- Combines (joins) the two DataFrames, `employee_sales` and `employee_info`, based on the common column "Employee ID." The result is stored in the `merged_df` DataFrame.

```
# 2. Create a new column for area code
def extract_area_code(phone):
    if pd.notna(phone):
        return int(phone.split("-")[0])
    else:
        return None
```

- Defines a function `extract_area_code` that takes a `phone` number as input and extracts the area code from it. It returns an integer area code if the phone number is not null; otherwise, it returns None.

```
merged_df["Area Code"] = merged_df["Phone"].apply(extract_area_code)
```

- Creates a new column called "Area Code" in the `merged_df` DataFrame by applying the `extract_area_code` function to the "Phone" column.

```
# 3. Calculate department performance and find the winning department
merged_df["Performance"] = merged_df["Sales"] - merged_df["Costs"]
```

- Calculates the performance of each employee by subtracting their "Costs" from their "Sales" and stores the result in a new column called "Performance" in the `merged_df` DataFrame.

```
department_performance = merged_df.groupby("Department Name")["Performance"].sum()
```

- Groups the data in the `merged_df` DataFrame by the "Department Name" column and calculates the sum of the "Performance" values within each group. The result is stored in the `department_performance` Series.

```
winner = department_performance.idxmax()
```

- Identifies the department with the highest total performance (maximum sum of "Performance" values) and assigns its name to the `winner` variable.

```
# Print the resulting DataFrame and the winning department
print(merged_df)
print("\nThe winning department is:", winner)
```

- Finally, it prints the `merged_df` DataFrame, which contains the combined employee data with area codes and performance, as well as the name of the winning department.

[ ]:
```
# 8. (10 points) You have a Pandas DataFrame named product_df with the␣
 ↪following columns: ProductID,
# ProductName, Category, Price, and InStock. The DataFrame contains information␣
 ↪about various products.

# (a) Create a new DataFrame named affordable_products_df containing only the␣
 ↪products with a price less than
# $50 and are currently in stock.

# (b) Calculate and create a new DataFrame named category_summary_df with two␣
 ↪columns: Cate- gory and Total
# Products. The Category column should contain unique product categories, and␣
 ↪the Total Products column should
# contain the count of products in each category.

#Helper code
# Import the Pandas library and alias it as 'pd' for convenience
import pandas as pd

# Sample data for the product_df DataFrame
data = {
    'ProductID': [1, 2, 3, 4, 5],
    'ProductName': ['Jacket', 'iPhone', 'Airpods', 'Lamp', 'Adapter'],
    'Category': ['Clothing', 'Electronics', 'Electronics', 'Home',␣
 ↪'Electronics'],
    'Price': [45, 60, 35, 25, 55],
    'InStock': [True, True, False, True, True]
}
# Create the product_df DataFrame using the sample data
product_df = pd.DataFrame(data)
```

[ ]:
```
#8a)
# - Filter products with a price less than 50 and in stock (InStock == True)
affordable_products_df = product_df[(product_df['Price'] < 50) &␣
 ↪(product_df['InStock'] == True)]
print("\nAffordable Products in stock:")
print(affordable_products_df)
```

[ ]:
```
#8b)

# - Group products by 'Category' and calculate the total count of products in␣
 ↪each category
category_summary_df = product_df.groupby('Category').size().
 ↪reset_index(name='Total Products')
print("\nCategory Summary:")
print(category_summary_df)
```

[ ]:
```
# Question 9

import requests
import json
import pandas as pd

response = requests.get('https://archive-api.open-meteo.com/v1/era5?latitude=37.
 ↪09&longitude=-95.
 ↪71&start_date=2000-01-01&end_date=2023-09-30&daily=temperature_2m_max,temperature_2m_min&ti
 ↪text
data = json.loads(response)
daily_temp_df = pd.DataFrame(data['daily'])

daily_temp_df.rename(columns={"temperature_2m_max": "max_temp",␣
 ↪"temperature_2m_min": "min_temp"}, inplace=True)
```

[ ]:
```
# Calculate the rolling average for a 7-day window (1 week)
weekly_avg_temp = daily_temp_df.rolling(window=7, step=7).mean().reset_index().
 ↪drop('index', axis=1)

# Now, 'weekly_avg_temp_df' contains the average weekly high and low␣
 ↪temperatures

# 2. Calculate if the average temperature in August is increasing each year
# Extract year and month from the 'Date' column
daily_temp_df['Month'] = daily_temp_df["time"].apply(lambda x: int(x.
 ↪split('-')[1]))
daily_temp_df['Year'] = daily_temp_df["time"].apply(lambda x: int(x.
 ↪split('-')[0]))

# Filter for August (Month == 8) and group by year
august_avg_temp_df = daily_temp_df[daily_temp_df['Month'] == 8].
 ↪groupby('Year')['max_temp'].mean()

# Check if the average temperature is increasing each year
prev = 0
result = True
```

```
for year in august_avg_temp_df:
    if year > prev:
        prev = year
    else:
        result = False
        break

# Print the result
print("Is the average temperature in August increasing each year?", result)
```

Here's a line-by-line explanation of the above code:

```
# Calculate the rolling average for a 7-day window (1 week)
weekly_avg_temp = daily_temp_df.rolling(window=7, step=7).mean().reset_index().drop('index', a
```

- This line calculates the rolling average of temperature data in a DataFrame called `daily_temp_df`.
- `.rolling(window=7, step=7)` specifies a rolling window of 7 days (1 week) and a step size of 7 days, which means it calculates the average for non-overlapping 7-day periods.
- `.mean()` computes the mean (average) temperature for each 7-day period.
- `.reset_index()` resets the index of the resulting DataFrame to the default integer index to get rid of the "daily" index
- `.drop('index', axis=1)` removes the old index column, leaving only the 'Date' and high avg and min avg columns in the `weekly_avg_temp` DataFrame.

```
# Extract year and month from the 'Date' column
daily_temp_df['Month'] = daily_temp_df["time"].apply(lambda x: int(x.split('-')[1]))
daily_temp_df['Year'] = daily_temp_df["time"].apply(lambda x: int(x.split('-')[0]))
```

- These lines extract the year and month from a 'Date' or 'time' column in the `daily_temp_df` DataFrame.
- The `lambda` functions split the date string by '-' and convert the extracted year and month to integers, then assign them to new 'Year' and 'Month' columns in the DataFrame.

```
# Filter for August (Month == 8) and group by year
august_avg_temp_df = daily_temp_df[daily_temp_df['Month'] == 8].groupby('Year')['max_temp'].mea
```

- This code filters the `daily_temp_df` DataFrame to select only the rows where 'Month' is equal to 8, which represents August.
- It then groups the filtered data by 'Year' and calculates the mean (average) of the 'max_temp' column for each year, storing the result in the `august_avg_temp_df` Series.

```
# Check if the average temperature is increasing each year
prev = 0
result = True
for year in august_avg_temp_df:
    if year > prev:
        prev = year
    else:
        result = False
        break
```

- This code checks if the average August temperature is increasing each year.
- It initializes `prev` to 0 to store the previous year's average temperature.
- It initializes `result` to `True`, assuming the temperatures are increasing unless proven otherwise.
- It iterates through each year's average temperature in `august_avg_temp_df`.
- If the current year's average temperature (`year`) is greater than the previous year's (`prev`), it updates `prev` with the current year's temperature.
- If at any point the current year's temperature is not greater than the previous year's, it sets `result` to `False` and breaks out of the loop early.

```
# Print the result
print("Is the average temperature in August increasing each year?", result)
```

- Finally, this line prints the result of whether the average temperature in August is increasing each year based on the value of the `result` variable.

[ ]:
```
# 10. You have a dataset containing the monthly sales figures (in dollars) for␣
 ↪a retail store over a year. Your task is to perform various operations using␣
 ↪NumPy.

# sales_data = [12000, 13500, 11000, 14500, 15000, 13200, 11800, 15200, 16200,
#   17500, 18000, 14000]

# (a) Create a NumPy array named sales_data from the dataset above.
# (b) Calculate and print the monthly growth rate for sales as a percentage.␣
 ↪The growth rate for a particular month is calculated as:\
# Growth Rate = ((Sales in Current Month - Sales in Previous Month)/Sales in␣
 ↪Previous Month) × 100
# (c) Find and print the month with the highest sales and the corresponding␣
 ↪sales amount.

#10a)

import numpy as np

sales_data = [12000, 13500, 11000, 14500, 15000, 13200, 11800, 15200, 16200,␣
 ↪17500, 18000, 14000]
sales_data = np.array(sales_data)
```

[ ]:
```
# 10b)
# Initialize an array for growth rates. Let the growth rate of first month be 0.
growth_rates = [0]

# Calculate the growth rate for each month starting from the second month␣
 ↪(index 1)
for i in range(1, len(sales_data)):
    previous_month_sales = sales_data[i - 1]
    current_month_sales = sales_data[i]
```

```
    growth_rate = ((current_month_sales - previous_month_sales) /␣
 ↪previous_month_sales) * 100
    growth_rates.append(growth_rate)

# Print the growth rates
enumMonths = ["January", "February", "March", "April", "May", "June", "July",␣
 ↪"August", "September", "October", "November", "December"]
for i, rate in enumerate(growth_rates):  # Start from month 2 (index 2)
    print(f"{enumMonths[i]}: Growth Rate = {rate:.2f}%")
```

[ ]:
```
#10c)

# Find the index of the month with the maximum sales:
max_sales_index = np.argmax(sales_data)

# Finally, print the month name and the corresponding sales amount.
print("Maximum sales were done in the month of", enumMonths[max_sales_index],␣
 ↪"for the amount of", sales_data[max_sales_index])
```