Name: _____
Surname: _____
Date: _____

Instructions

Always assume each code is run independently, so variables do not persist between exercises.

Questions 1-6: Multiple Choice (5 points each)

On the table below, mark the correct answer for each question. Use an X to mark your choice. If you want to change your answer, fill the entire box and mark a new one. If multiple boxes are marked for the same question, no points will be awarded. See the examples below. Each question has **only one** correct answer. A correct answer awards 5 points. There is no penalty for wrong choices.

Example	A	B	C	D
Chosen A	X	☐	☐	☐
Chosen C, cancelled A	■	☐	X	☐
Chosen B, cancelled A and C	■	X	■	☐
Invalid (multiple choices)	X	X	☐	☐
Invalid (no choices)	■	☐	☐	☐

Questions 7-9: Debugging (10 points each)

These exercises show code cells with one or more errors. You have to find these errors, explain why they happen, and suggest a fix to them.

Questions 10-11: Open Problems (20 points each)

These are open-ended tasks. Explain your approach, outline algorithmic steps, and optionally provide compact working code snippets.

Mark your answers for Questions 1–6 below

Question	A	B	C	D
Q1	☐	☐	☐	☐
Q2	☐	☐	☐	☐
Q3	☐	☐	☐	☐
Q4	☐	☐	☐	☐
Q5	☐	☐	☐	☐
Q6	☐	☐	☐	☐

1. Which statement about Python function arguments is FALSE?

    A. You can mix positional and keyword arguments as long as positionals come first.

    B. Mutable default arguments retain changes between calls.

    C. Type hints enforce runtime types.

    D. You can define a function that uses only keyword arguments.

2. Which of the following type conversions will raise an error?

    A. `int("3.14")`

    B. `float("2.71" + "3")`

    C. `str(42 / 7)`

    D. `bool(1 + 0)`

3. Which of the following lines will raise an error?

```
sports = ("football", "chess", ["dancing"])
sports[2].append("basketball")   # Line 2
print(f"I love {sports[-1]}")      # Line 3
print(sports[0][4])                 # Line 4
```

    A. Line 2.

    B. Line 3.

    C. Line 4.

    D. None of them.

4. What type of input will the following function accept without raising an error?

```
def find_three_unique_highest(scores):
    unique_scores = set(scores)
    sorted_scores = sorted(unique_scores, reverse=True)
    return sorted_scores[:3]
```

    A. A mixed list of strings and integers.

    B. A tuple of floats.

    C. A 2D numpy array of numbers.

    D. All of the other three.

5. Which of the following statements about Numpy arrays is FALSE?

   A. Numpy arrays can only contain elements of the same data type.

   B. You can perform element-wise operations on Numpy arrays of the same shape.

   C. Numpy arrays can be initialized from Python lists or tuples.

   D. Numpy arrays do not support slicing operations.

6. What will be the output of the following code snippet?

```
import numpy as np

matrix = np.array([
    [1, 2, 3],
    [4, 5, 6]])

for row in matrix:
    if row > 2:
        print(row[row > 2])
```

   A. `[False False True]` and then `[True True True]`.

   B. `[3]` and then `[4 5 6]`.

   C. The code runs fine but produces no output.

   D. An error is raised.

Correct answers for Questions 1–6:

Question	Key
1	C
2	A
3	D
4	B
5	D
6	D

7. The following code is intended to capitalize each name in the list and then remove the lowercase versions. However, the execution never ends. Identify the one problem and suggest a fix.

```
names = ["john", "Rob", "arya", "Sansa"]

# We want to capitalize each name
for n in names:
    new = n.capitalize()  # Uppercases first letter
    names.append(new)  # Adds the element to the end of the list

# Now remove the lowercase versions
idx = 0
while idx < len(names):
    n = names[idx]
    if n.islower():  # True if all letters are lowercase
        names.remove(n)  # Removes the element from the list
    else:
        idx += 1
```

Answer:

The problem arises because we are modifying the list `names` while iterating over it in the `for` loop. Specifically, we are appending new capitalized names to the list, which causes the loop to never terminate as the length of the list keeps increasing. *(5 points)*

To fix this, we can create a new list to store the capitalized names instead of modifying the original list during iteration. *(5 points)*

Here is the corrected code:

```
names = ["john", "Rob", "arya", "Sansa"]
new_names = []  # Create a new list to store capitalized names

# We want to capitalize each name
for n in names:
    new = n.capitalize()  # Uppercases first letter
    new_names.append(new)  # Adds the element to the end of the list

# With this change, the while loop is no longer necessary
# as we have already created a new list with capitalized names.
```

8. The function below should extract the antidiagonal of any square matrix. The code shows abnormal behaviour: it returns incorrect results for square matrices. Identify the one problem and suggest a fix.

```
import numpy as np


def antidiagonal(x: np.array) -> np.array:
    if x.ndim != 2:
        print("Array is not 2D")
    elif x.shape[0] != x.shape[1]:
        print("Matrix is not squared")
    else:
        # The antidiagonal row indices are the same as the row indices
        row_indices = np.arange(x.shape[0])
        # The antidiagonal column indices are the reverse of the row
    indices
        col_indices = -row_indices
        return x[row_indices, col_indices]


a = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]])
print(antidiagonal(a))
# Expected: [3 5 7]
# Output: [1 6 8]
```

Answer:

The problem lies in the way the column indices for the antidiagonal are calculated. Using negative indexing with `-row_indices` does not yield the correct column indices for the antidiagonal. Instead, it results in incorrect positions being accessed in the array. *(5 points)*

To fix this, we should calculate the column indices as `-row_indices - 1`, which correctly maps the row indices to their corresponding antidiagonal column indices. *(5 points)*

Here is the corrected code:

```
import numpy as np


def antidiagonal(x: np.array) -> np.array:
    if x.ndim != 2:
        print("Array is not 2D")
    elif x.shape[0] != x.shape[1]:
        print("Matrix is not squared")
    else:
        # The antidiagonal row indices are the same as the row indices
        row_indices = np.arange(x.shape[0])
        # The antidiagonal column indices are the reverse of the row
    indices
        col_indices = -row_indices - 1
        return x[row_indices, col_indices]
```

9. The following code is intended to simulate two different random walks. However, both arrays are identical, no matter the `seed` we use. Identify the two problems in this code (one of which is related to the issue we mentioned), and suggest a fix.

```
import numpy as np

seed = 42 # Can be any other number
np.random.seed(seed)  # Set seed for reproducibility

# Generate two random arrays of 0's and 1's
walk1 = np.random.randint(0, 1, size=1000)
walk2 = np.random.randint(0, 1, size=1000)

# Change 0's to -1's to represent backward steps
mask1 = (walk1 == 0)
mask2 = (walk2 == 0)
walk1[mask2] = -1
walk2[mask1] = -1

print(np.all(walk1 == walk2))
# Expected: False (arrays are different)
# Output: True (arrays are identical)
```

Answer:

The first problem is with the parameters passed to `np.random.randint()`. The function generates random integers in the range `[low, high)`, meaning that the `high` value is exclusive. Therefore, using `np.random.randint(0, 1, size=1000)` will only generate 0's, resulting in identical arrays. *(5 points)*

The second problem is that the masks used to change 0's to -1's are incorrectly applied. The masks `mask1` and `mask2` are swapped when modifying `walk1` and `walk2`, leading to both arrays being modified in the same way and thus remaining identical. *(5 points)*

Here is the corrected code:

```
import numpy as np

seed = 42 # Can be any other number
np.random.seed(seed)  # Set seed for reproducibility

# Generate two random arrays of 0's and 1's
walk1 = np.random.randint(0, 2, size=1000)
walk2 = np.random.randint(0, 2, size=1000)

# Change 0's to -1's to represent backward steps
mask1 = (walk1 == 0)
mask2 = (walk2 == 0)
walk1[mask1] = -1
walk2[mask2] = -1
```

10. In machine learning, it is very common to normalize the input data. One common normalization technique is min-max scaling, which transforms each feature to a given range, typically [0, 1]. The formula for min-max scaling is:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

where X is the original feature value, X_{min} and X_{max} are the minimum and maximum values of the feature in the dataset, and X' is the normalized value.

Because neural networks use multiple features, the input data is often represented as a 2D array (matrix), where each row corresponds to a data point and each column corresponds to a feature. In that case, the min-max scaling should be applied independently to each feature (i.e., each column).

Describe how you would implement min-max scaling for a 2D array in Python.

```
example_input = np.array([
    [10, 200, 0.5],
    [20, 300, 0.75],
    [15, 250, 0.6],
    # can be many more rows
])  # 2D array where each row is a data point and each column is a
   feature

expected_output = np.array([
    [0.0, 0.0, 0.0],
    [1.0, 1.0, 1.0],
    [0.5, 0.5, 0.3333],
    # normalized values
])  # 2D array with normalized features
```

Answer:

We want to implement a function in the form:

```
def min_max_scaling(x: np.array) -> np.array
```

that takes a 2D numpy array as input and returns (not prints!) a new 2D numpy array with the min-max scaled features. *(4 points)*

The min-max operation is applied to each column, which means we will be using `axis=0` when calculating the minimum and maximum values. *(4 points)* In code:

```
xmin = x.min(axis=0)  # Minimum value for each feature (column)
xmax = x.max(axis=0)  # Maximum value for each feature (column)
```

And that will give us two 1D arrays with the minimum and maximum values for each feature. *(2 points)*

Then, we can use broadcasting to apply the min-max scaling formula to the entire 2D array at once: *(10 points)*

```
x_normalized = (x - xmin) / (xmax - xmin)
```

Alternatively, we could use a loop to iterate over each column and apply the formula individually. *(6 points instead of 10 points for broadcasting)*

And that will give us the desired output, a 2D array with the normalized features.

Full function implementation:

```
def min_max_scaling(x: np.array) -> np.array:
    xmin = x.min(axis=0)  # Minimum value for each feature (column)
    xmax = x.max(axis=0)  # Maximum value for each feature (column)
    x_normalized = (x - xmin) / (xmax - xmin)
    return x_normalized
```

11. You are given a list of tuples `(transaction_date, customer_id, transaction_amount)`. Compute the total amount spent by each customer. Once you have the totals, find a fast way to see the spending of any customer by using their ID. Finally, identify the customer with the highest total.

    Describe your approach and outline the steps you would take in Python.

```python
example_input = [
    ("2023-01-01", "C001", 150.0),
    ("2023-01-02", "C002", 200.0),
    ("2023-01-03", "C001", 100.0),
    ("2023-01-04", "C003", 250.0),
    ("2023-01-05", "C002", 300.0),
    # can be many more transactions
]  # (transaction_date, customer_id, transaction_amount)

# Output 1: total spent by each customer
# Output 2: given any customer_id, return their total spent
# Output 3: customer_id with highest total spent
```

Answer:

Because we need to compute totals for each customer and then quickly access these totals by customer ID, a dictionary is an ideal data structure for this task. *(5 points)*

We can define a function in the form:

```python
def compute_customer_spending(transactions: list) -> dict
```

that takes a list of transactions as input and returns a dictionary where the keys are customer IDs and the values are their total spending. *(5 points, alternative to previous answer)*

We can iterate over the list of transactions, and for each transaction, we extract the customer ID and transaction amount. We then update the total spending for that customer in the dictionary. If the customer ID is not already in the dictionary, we initialize their total to zero before adding the transaction amount. *(5 points)*

Here is the implementation:

```python
def compute_customer_spending(transactions: list) -> dict:
    customer_totals = {}
    for date, customer_id, amount in transactions:
        if customer_id not in customer_totals:
            customer_totals[customer_id] = 0.0
        customer_totals[customer_id] += amount
    return customer_totals

# Example usage
customer_totals = compute_customer_spending(example_input)
```

Now that we have the total spending for each customer in a dictionary, we can easily access the total spent by any customer using their ID. *(5 points)* For example:

```python
customer_id = "C002"
total_spent = customer_totals.get(customer_id, 0.0)
# Returns 500.0 for C002 in the example input
```

Finally, to identify the customer with the highest total spending, we can use the `max()` function with a custom key that retrieves the total spending from the dictionary. *(5 points)*

```
highest_spender = max(customer_totals, key=customer_totals.get)
print(f"Customer with highest total spent: {highest_spender} with amount
    {customer_totals[highest_spender]}")
```

Alternatively, we could iterate through the dictionary items to find the maximum total spending. *(5 points instead of 5 points for using `max()`)*

```
highest_spender = None
max_spent = 0.0
for customer_id, total in customer_totals.items():
    if total > max_spent:
        max_spent = total
        highest_spender = customer_id
print(f"Customer with highest total spent: {highest_spender} with amount
    {max_spent}")
```