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)

Circle the correct answer for each question, as each question has **only one** correct answer. A correct answer awards 5 points. There is no penalty for wrong choices.

Questions 7-9: Debugging (10 points each)

The functions below contain 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)

In the following exercises, you will encounter problems that are more open-ended and may not have a single correct answer. Your task is to explain how you would approach each problem by breaking it down into smaller, manageable steps.

Describe how you would structure your code, which logical steps are needed, and what kind of Python tools or constructs (e.g., loops, conditionals, functions, lists, etc.) you would use. You are not required to write full code, but doing so is encouraged if it helps clarify your ideas.

---

1. Which of these statements is **TRUE** about **integer indexing** with arrays?

   (a) Integer indexing allows selecting arbitrary elements of an array using a list or array of integer positions.

   (b) Integer indexing can only be used when the array is sorted.

   (c) Integer indexing can only be applied to one–dimensional arrays.

   (d) Integer indexing modifies the original array in place.

2. After running the following code, what will be the value of "str2"?

```
str1 = "Merry Christmas"

str2 = ""
vowels = "aeiou"
for char in str1:
    if char in vowels:
        continue
    elif char == " ":
        str2 += "-"
    else:
        str2 += char.lower()
```

   (a) `str2 = ''M''`

   (b) `str2 = ''mrry-chrstms''`

   (c) `str2 = ''merry-christmas''`

   (d) The code will raise an error

3. Which of the following statements about mutability in Python is **FALSE**?

   (a) Modifying a list inside a function also modifies it outside the function, even without returning it.

   (b) If we define a default parameter for a function as an empty list `[]`, a new empty list is created on every function call.

   (c) We can modify a list that contains tuples, but we cannot modify the tuples inside the list.

   (d) To create an independent copy of a list, assigning it to another variable is not enough; we must use `.copy()` or another copying method.

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

```
coordinates = ((0, 0), (0,1), (1, 0), (1, 1))
new_coord = []
new_coord.append(coordinates[2])    #Line 3
print(f"Last coordinate {coordinates[-1]}")      #Line 4
coordinates[0] = (coordinates[0][0] - 1, coordinates[0][1] - 1) #Line 5
```

   (a) Line 3.

   (b) Line 4.

   (c) Line 5.

   (d) None of them.

5. After running this code, what will be the final value of "common_holidays"?

```
holidays1 = [1, 4, 12, 13, 14, 17, 23, 24]
holidays2 = [4, 5, 6, 7, 24, 25, 26, 31]
holidays3 = [1, 2, 3, 24, 26, 27, 28]

common_holidays = set(holidays1) & set(holidays2) - set(holidays3)

common_holidays = [str(day) + " of December" for day in common_holidays]

print(common_holidays)
```

(a) `common_holidays = ["24 of December", "4 of December"]`

(b) `common_holidays = ["4 of December"]`

(c) `common_holidays = [4, "of December"]`

(d) The code will raise an error.

6. What will be the value of array `B` after applying the function `manipulate_arr()`?

```
import numpy as np

def manipulate_arr(arr: np.ndarray)->np.ndarray:

    C = np.arange(3) + 10

    D = np.hstack((arr, C.reshape(C.size, 1)))

    E = np.vstack((D, C))

    return E

A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

B = manipulate_arr(A)
```

(a) $\begin{pmatrix} 1 & 2 & 3 & 10 \\ 4 & 5 & 6 & 10 \\ 7 & 8 & 9 & 10 \\ 10 & 10 & 10 & 10 \end{pmatrix}$

(b) $\begin{pmatrix} 1 & 2 & 3 & 10 \\ 4 & 5 & 6 & 11 \\ 7 & 8 & 9 & 12 \end{pmatrix}$

(c) $\begin{pmatrix} 1 & 2 & 3 & 10 \\ 4 & 5 & 6 & 11 \\ 7 & 8 & 9 & 12 \\ 10 & 11 & 12 & 0 \end{pmatrix}$

(d) The code will raise an error because shapes do not match.

Answers: 1-a, 2-b, 3-b, 4-c, 5-b, 6-d

7. The following code is intended to decrypt messages that were encrypted as follows: wherever the original message contains a space, a dollar sign `$` is placed instead, and extra spaces are inserted between words where there were none. To decrypt it, the `$` symbols should be replaced by spaces and the extra spaces removed. However, the code does not produce the desired output. For example, it returns a list instead of a string. Identify the four errors and correct them.

```
1  def decrypt_msg(message: str)->str:
2    #Split message without "$"
3    message_list = message.split(" ")
4
5    #Iterate over message and replace $
6    decrypt_msg = []
7    for word in message_list:
8      word.replace("$", "")
9      decrypt_msg.append(word)
10
11   return decrypt_msg
12
13 input_message = "I$a m$ d oi ng$ a$ g r ea t$ex am"
14
15 print(decrypt_msg(input_message))
16
17 #Expected output
18 #"I am doing a great exam"
19
20 #Actual output
21 #['I$a', 'm$', 'd', 'oi', 'ng$', 'a$', 'g', 'r', 'ea', 't$ex', 'am']
```

Answer:

The code has four errors. First, in line 8 the result of `word.replace("$", "")` is never stored. Strings are immutable, so `replace` does not modify `word`: it returns a new string that is thrown away here. We need to assign it, `word = word.replace(...)`, or directly append the result. *(+3 points)*

Second, the replacement string is `""`, which deletes each dollar sign. According to the encryption rule, every `$` was a space in the original message, so it must be replaced by a space `" "`. *(+2 points)*

Third, the function returns `decrypt_msg`, which is a list, while the function is supposed to return a string. After splitting by spaces, the extra spaces are already gone, so we must glue the pieces together **without** any separator: `"".join(...)`. Joining with a space would reintroduce the extra spaces. *(+3 points)*

Fourth, the local variable `decrypt_msg` in line 6 has the same name as the function itself. It does not crash here, but it shadows the function name inside its own body, which is a bad practice (for instance, a recursive call would fail). We should rename it, e.g. to `decrypted_words`. *(+2 points)*

(Optional, fixed code)

```
1  def decrypt_msg(message: str) -> str:
2      # Split message by the extra spaces
3      message_list = message.split(" ")
4
```

```
    # Iterate over message and replace $ by a space
    decrypted_words = []
    for word in message_list:
        word = word.replace("$", " ")
        decrypted_words.append(word)

    # Join the pieces without any separator
    return "".join(decrypted_words)
```

8. This code is intended to simulate the random walks of two particles in two dimensions with the same number of steps and compare their final positions. The function raises an error on line 14: "`ValueError: The truth value of an array with more than one element is ambiguous`". Identify the two errors and correct them.

```
1  import numpy as np
2
3  def random_walk_2D(directions: list[tuple], num_steps: int)->bool:
4    #Generate two random arrays of directions
5    pos_arr1 = rng.choice(directions, size = num_steps)
6    pos_arr2 = rng.choice(directions)
7
8    #Calculate the final position [posx, posy] of each particle
9    #computing the cumsum of each column
10   final_pos1 = np.cumsum(pos_arr1, axis = 0)
11   final_pos2 = np.cumsum(pos_arr2, axis = 0)
12
13   #Compare the final position
14   if final_pos1[-1] == final_pos2[-1]:
15     return True
16
17   else:
18     return False
19
20 #Create list of directions
21 directions = [(0, 0), (0, 1), (1, 0), (0, -1), (-1, 0)]
22
23 rng = np.random.default_rng(40)
24
25 random_walk_2D(directions, 10)
```

Answer:

The first error is in line 6: `rng.choice(directions)` is missing the `size` argument, so it returns a single random direction (an array with two numbers) instead of `num_steps` directions. As a consequence, the `np.cumsum` of line 11 returns the cumulative sum of the two coordinates, and `final_pos2[-1]` is a single number instead of the final position `[posx, posy]`. Both particles need the same number of steps, so line 6 must be `rng.choice(directions, size=num_steps)`. *(+5 points)*

The second error is the comparison in line 14. The expression `final_pos1[-1] == final_pos2[-1]` compares two arrays element by element, which returns a Boolean array (one value per coordinate), and an `if` statement needs a single Boolean value. This is the origin of the "ambiguous truth value" error. We need to reduce that array to one Boolean: `np.all(final_pos1[-1] == final_pos2[-1])` or `np.array_equal(final_pos1[-1], final_pos2[` *(+5 points)*

(Optional, fixed code)

```
1  import numpy as np
2
3  def random_walk_2D(directions: list[tuple], num_steps: int) -> bool:
4      # Generate two random arrays of directions
5      pos_arr1 = rng.choice(directions, size=num_steps)
6      pos_arr2 = rng.choice(directions, size=num_steps)
7
```

```
    # Calculate the final position [posx, posy] of each particle
    # computing the cumsum of each column
    final_pos1 = np.cumsum(pos_arr1, axis=0)
    final_pos2 = np.cumsum(pos_arr2, axis=0)

    # Compare the final position
    if np.all(final_pos1[-1] == final_pos2[-1]):
        return True
    else:
        return False
```

9. This code is intended to create a 2-D array A from an array B with 2 columns and an odd number of rows, placing the two columns of B into A in the shape of a cross (see the expected output). The function raises an error on line 15: "`ValueError: too many indices for array: array is 1-dimensional, but 2 were indexed`". Identify the three errors and correct them.

```
1  import numpy as np
2
3  def cross_matrix(B: np.ndarray)->np.ndarray:
4    #Get number of rows
5    n = B.shape[1]
6
7    #Create an array filled with zeros
8    A = np.zeros(n)
9
10   #Extract both columns
11   column1 = B[0, :]
12   column2 = B[:, 1]
13
14   #Add the columns as a cross to the new array
15   A[:, n//2] = column1
16   A[n//2, :] = column2
17
18   return A
19
20 B = np.array([[1, 4],
21                [2, 5],
22                [3, 6]])
23
24 print(f"Cross array:\n {cross_matrix(B)}")
25
26 #Expected output
27 #Cross array:
28 #[[0. 1. 0.]
29 #[4. 5. 6.]
30 #[0. 3. 0.]]
```

Answer:

The first error is on line 5, where we get the number of rows with `B.shape[1]`. This actually gives us the number of columns. The correct way to get the number of rows is `n = B.shape[0]`. *(3 points)*

The second error is on line 8, where we create the array `A` with `np.zeros(n)`. This creates a 1-D array of length `n`, but we need a 2-D array of shape `(n, n)` to place the columns in a cross shape. The correct code is `A = np.zeros((n, n))`. This is the origin of the error message, as a 1-D array cannot be indexed with two indices. *(3 points)*

The third error is the indexing on line 11. The expression `B[0, :]` accesses the first row of B, but we want the first column. The correct way to get the first column is `B[:, 0]`. The second column, `B[:, 1]`, is already correct. *(4 points)*

(Optional, fixed code)

```
1  import numpy as np
2
3  def cross_matrix(B: np.ndarray) -> np.ndarray:
```

```
    # Get number of rows
    n = B.shape[0]

    # Create an array filled with zeros
    A = np.zeros((n, n))

    # Extract both columns
    column1 = B[:, 0]
    column2 = B[:, 1]

    # Add the columns as a cross to the new array
    A[:, n//2] = column1
    A[n//2, :] = column2

    return A
```

10. In engineering applications, it is common to work with sensors that report precision along three axes (X, Y, and Z). Over time, sensors lose precision on all axes, and only those whose precision remains above a required threshold should be kept.

    In this setting, the data is represented as a 2D array (matrix), where each row corresponds to a sensor and each column corresponds to its precision on one axis. Each year, every sensor loses a random amount of precision between 0 and 0.2, and this loss is applied to all three axes of that sensor. The yearly precision update for each value can be expressed as:

    $$\text{sensor_precision}' = \text{sensor_precision} \cdot (1 - \text{precision_loss})$$

    where precision_loss contains the randomly generated precision loss for each sensor.

    After applying this yearly loss, sensors whose precision on all three axes remains strictly above 70% are kept, whereas any sensor that falls below this threshold has its entire row replaced with zeros. This degradation-and-filtering process is repeated each year.

    Create a function called `filter_sensors_by_precision()`. It takes three inputs:

    - `sensor_precision`: a 2D NumPy array where each row represents a sensor and each column represents its precision on one axis.
    - `minimum_precision`: the threshold below which a sensor is discarded.
    - `years`: the number of consecutive years over which precision degradation is applied.

    The function must return a 2D NumPy array in which sensors that remain above the threshold on all three axes keep their updated precision values, while sensors that fall below the threshold (**in any axis**) have their entire row set to 0.

    If all sensors fall below the threshold, the function should print a message indicating that no sensors maintain the required precision after the given period.

```
sensor_precision = np.array([
    [96.85, 95.25, 85.47],
    [97.43, 92.12, 95.48],
    [86.32, 90.05, 75.41],
    [90.12, 85.15, 80.00],
])  # each row is a sensor, columns are precisions on X, Y, Z

possible_expected_output = np.array([
    [0, 0, 0],
    [0, 0, 0],
    [65.30516806, 68.12708971, 57.05123637],
    [0, 0, 0]
])
```

Answer:

The function receives a 2-D array, a threshold and a number of years. Because the process must be repeated year after year, we use a for loop that runs `years` times, `for year in range(years)`. Before the loop, we work on a float copy of the input, `precision = sensor_precision.astype(float)`, so that we do not modify the array of the user. *(+4 points)*

Inside the loop we generate one random loss per sensor (not per value), because the same loss is applied to the three axes of a sensor. We use `rng.uniform(0, 0.2, size=(n_sensors,`

`1))`, which creates a column vector with one number per row. Thanks to broadcasting, the update `precision = precision * (1 - loss)` multiplies each row by its own factor. *(+6 points)*

Next, we build a Boolean mask with one value per sensor: `keep = np.all(precision > minimum_precision, axis=1)`. The `axis=1` makes `np.all` check the three columns of each row, so a single axis below the threshold discards the sensor. Then we set the entire discarded rows to zero with `precision[~keep] = 0`. Discarded sensors stay at zero in the following years, as they cannot go back above the threshold. *(+6 points)*

After the loop, we check if any sensor survives with `np.any(precision)`. If it is False, we print the message saying that no sensor maintains the required precision. Finally, we return the array. *(+4 points)*

Comment: The statement mentions a 70% threshold, but the function receives it as `minimum_precision`, so we use that parameter. As the loss is random, the output changes from run to run unless we fix the seed of the generator.

(Optional, code)

```python
import numpy as np

def filter_sensors_by_precision(sensor_precision: np.ndarray,
                                minimum_precision: float,
                                years: int) -> np.ndarray:
    rng = np.random.default_rng()
    # Work on a float copy, so the input is not modified
    precision = sensor_precision.astype(float)
    n_sensors = precision.shape[0]

    for year in range(years):
        # One random loss per sensor, as a column to use broadcasting
        loss = rng.uniform(0, 0.2, size=(n_sensors, 1))
        precision = precision * (1 - loss)

        # A sensor is kept only if all its axes are above the threshold
        keep = np.all(precision > minimum_precision, axis=1)
        # Discarded sensors have the whole row set to zero
        precision[~keep] = 0

    # If every value is zero, no sensor survived
    if not np.any(precision):
        print("No sensors maintain the required precision after", years,
    "years.")

    return precision
```

11. You are a programmer who wants to create a task-management tool. The starting point is a weekly task schedule stored as a nested list. Each element contains a day followed by tuples of the form `(hour, task)`. (**See the example below**)

    To begin, you will create a function called `tasks_list_to_dict()` that allows direct access to the tasks of each day. The function receives the nested list `tasks_week` as input and converts this structure into a dictionary of dictionaries, where the keys are day names and the values are dictionaries mapping hours to tasks.

    After that, you want to be able to add new tasks while checking whether a task already exists at the given time. To do this, ask the user for the day, hour, and task they want to add.

    Then create another function, `add_task()`, which checks whether there is already a task in that time slot: if there is, it prints a message saying that a task already exists at that hour; if not, it adds the new task to the dictionary.

    The function `add_task` takes the following inputs:

    - `task_dict`: The dictionary produced by `tasks_list_to_dict()`
    - `day`: The day entered by the user as a string
    - `hour`: The hour entered by the user as a string
    - `task`: The task entered by the user as a string

```
example_input = [
    ["Monday", ("09:00", "Check emails"), ("11:00", "Team meeting")],
    ["Tuesday", ("08:30", "Exercise"), ("10:00", "Write report")],
    ["Wednesday", ("09:00", "Team session"), ("12:00", "Work lunch")],
    ["Thursday", ("09:30", "Documentation"), ("13:00", "Meeting")],
    ["Friday", ("08:00", "New planning"), ("11:30", "Review report")]
]


#Output 1: Dictionary of dictionaries containing the tasks scheduled for each day and hour.
#Output 2: Dictionary updated with the newly added tasks (or a warning message if a task already exists at that hour).
```

Answer:

We need two functions. For `tasks_list_to_dict()`, the input is the nested list, and the output will be a dictionary of dictionaries. We start with an empty dictionary `task_dict = {}` and loop through the list, `for entry in tasks_week`. In each `entry`, the first element `entry[0]` is the day, and the rest of the elements `entry[1:]` are tuples `(hour, task)`. *(+5 points)*

For each day, we create an inner dictionary, `task_dict[day] = {}`, and then use a nested for loop over `entry[1:]`. Thanks to tuple unpacking, `for hour, task in entry[1:]` gives us the hour and the task directly, and we store them with `task_dict[day][hour] = task`. At the end, we return `task_dict`. *(+5 points)*

For `add_task()`, we use an if-else clause. If the `hour` is already a key in the dictionary of that day, `hour in task_dict[day]`, we print the message that a task already exists at

that hour. Otherwise, we add the new task with `task_dict[day][hour] = task`. As a dictionary is mutable, it is modified in place, but we can also return it. It is safer to first check that the day exists, and create it with an empty dictionary if not. *(+6 points)*

Finally, the main code asks the user for the three values using `input()`, which always returns strings, and calls both functions. *(+4 points)*

(Optional, code)

```
def tasks_list_to_dict(tasks_week: list) -> dict:
    task_dict = {}
    for entry in tasks_week:
        day = entry[0]
        task_dict[day] = {}
        # The rest of the elements are (hour, task) tuples
        for hour, task in entry[1:]:
            task_dict[day][hour] = task
    return task_dict


def add_task(task_dict: dict, day: str, hour: str, task: str) -> dict:
    # Create the day if it does not exist yet
    if day not in task_dict:
        task_dict[day] = {}

    if hour in task_dict[day]:
        print(f"There is already a task on {day} at {hour}: {task_dict[
    day][hour]}")
    else:
        task_dict[day][hour] = task
    return task_dict


# Main code
task_dict = tasks_list_to_dict(example_input)
day = input("Day: ")
hour = input("Hour (HH:MM): ")
task = input("Task: ")
task_dict = add_task(task_dict, day, hour, task)
print(task_dict)
```

Cheatsheet

Given a number named `x`:

- `x % y` returns the remainder of the division $\frac{x}{y}$.
- `x // y` returns the quotient of the division $\frac{x}{y}$.
- `abs(x)` returns the absolute value of `x`.

Given a string named `s`:

- `int(s)` returns an integer. Fails if `s` has invalid characters.
- `float(s)` returns a floating number. Fails if `s` has invalid characters.
- `s.lower()` returns a copy of the string, all in lowercase.
- `s.split(" ")` split a string into a list where each word is a list item.
- `s.count(<pattern>)` returns the number of non-overlapping occurrences of `<pattern>`.
- `s.find(<pattern>, <start>)` finds the first occurrence of the specified `<pattern>`, starting from index `<start>`. Returns `-1` if the pattern is not found.
- `s.replace(<pattern>, <new>)` returns a new string where all instances of `<pattern>` have been replaced with `<new>`.

Given a list named `ls`:

- `ls.append(element)` add element to that list.

Given a list or tuple named `ls`:

- `ls.index(element)` returns the index of element in that list.
- `len(ls)` returns the number of elements in that list.
- `sum(ls)` returns the sum of the elements of that list.
- `sorted(ls)` returns a sorted version of that list.
- `tuple(ls)` / `set(ls)` returns a tuple / set using the elements of that list.
- `", ".join(ls)` returns a string with the elements of the list separated by commas.

Given a set named `a`:

- `a.add(x)` adds the element `x` to the set `a`.
- `a.union(b)` or (`a | b`) returns the union of set `a` with another set `b`.
- `a.intersection(b)` or (`a & b`) returns the intersection of set `a` with another set `b`.
- `a.difference(b)` or (`a - b`) returns the difference of set `a` with another set `b`.

Given a dictionary named `d`:

- `list(d.keys())` returns a list with the keys of `d`.
- `list(d.values())` returns a list with the values of `d`.
- `list(d.items())` returns a list with the tuples (key, value) of `d`.

Some useful build-in functions:

- `type(x)` returns the type of `x` (int, float, list...).
- `range(start, stop)` generates numbers from `start` until `stop`, not including `stop`.
- `input(msg)` is used to take user input, in form of a string.

NumPy Cheatsheet

Importing NumPy:

- `import numpy as np`
- `rng = np.random.default_rng()`

Creating arrays:

- `np.zeros((m, n))` creates an $m \times n$ array filled with zeros.
- `np.ones((m, n))` creates an $m \times n$ array filled with ones.
- `np.arange(start, stop, step)` creates values from `start` to `stop` (exclusive) with a given `step`.
- `np.linspace(start, stop, num)` creates `num` evenly spaced values between `start` and `stop`.

Given a NumPy array `arr`:

- `arr.ndim` returns the number of array dimensions, as a number.
- `arr.shape` returns the shape of an array, as a tuple.

Array operations:

- `np.dot(a, b)` computes the dot product.
- `np.matmul(a, b)` or `a @ b` performs matrix multiplication.
- `np.transpose(a)` or `a.T` transposes the array.

Array manipulation:

- `a.reshape((m, n))` changes the shape of an array.
- `a.flatten()` flattens a multidimensional array into 1D.
- `np.concatenate((a, b), axis=0)` joins arrays along an axis.

- `np.stack(<sequence>)` joins a sequence of arrays along a new axis.
- `np.vstack((a, b))`, `np.hstack((a, b))` vertical/horizontal stacking.
- `np.split(a, n)` splits an array into `n` parts.

Statistics and descriptive analysis:

- `np.mean(a)` mean of all elements.
- `np.median(a)` median of all elements.
- `np.std(a)` standard deviation.
- `np.var(a)` variance.
- `np.min(a)`, `np.max(a)` minimum and maximum.
- `np.argmin(a)`, `np.argmax(a)` indices of minimum and maximum.
- `np.corrcoef(a, b)` correlation coefficient matrix.
- `np.cov(a, b)` covariance matrix.

Linear algebra:

- `np.linalg.inv(a)` inverse of matrix `a`.
- `np.linalg.det(a)` determinant of `a`.
- `np.linalg.eig(a)` eigenvalues and eigenvectors.
- `np.linalg.norm(a)` vector or matrix norm.
- `np.linalg.solve(a, b)` solves $Ax = b$.

Logical operations and conditions:

- `np.where(cond, x, y)` selects elements from `x` or `y` based on condition.
- `np.all(a)`, `np.any(a)` check if all/any elements are True.
- `np.unique(a)` returns sorted unique elements.

Random number generation (using `rng`):

- `rng.random(size)` uniform distribution in $[0, 1)$.
- `rng.integers(low, high, size)` random integers in $[low, high)$.
- `rng.normal(mean, std, size)` normal (Gaussian) distribution.
- `rng.uniform(low, high, size)` uniform distribution in $[low, high)$.
- `rng.choice(a, size)` random sample from array `a`.

Random number generation (legacy API):

- `np.random.randint(low, high, size)` returns random integers from low (inclusive) to high (exclusive).
- `np.random.uniform(low, high, size)` samples from a uniform distribution.
- `np.random.seed(seed)` fixes the seed. After a seed reset, the same set of numbers will appear every time.

Math and trigonometry:

- `np.power(arr1, arr2)` returns the first array elements raised to powers from second array, element-wise.
- `np.sin(arr)` returns the trigonometric sine, assuming angle in radians.
- `np.cos(arr)` returns the trigonometric cosine, assuming angle in radians.
- `np.radians(arr)` converts angles from degrees to radians.
- `np.degrees(arr)` converts angles from radians to degrees.