GROUP WORK PROJECT # 01 MScFE 600: FINANCIAL DATA
GROUP NUMBER: 2791

FULL LEGAL NAME	LOCATION (COUNTRY)	EMAIL ADDRESS	MARK X FOR ANY NON-CONTRIBUTING MEMBER
Marin Yordanov Stoyanov	Bulgaria	azonealerts@gmx.com	
Chia-Cheng Chang	Taiwan	xiong7238@gmail.com	
Azamat Zhaksylykov	USA	a.zhaksylykov@gmail.com	

Statement of integrity: By typing the names of all group members in the text boxes below, you confirm that the assignment submitted is original work produced by the group (excluding any non-contributing members identified with an "X" above).

Team member 1	**Marin Yordanov Stoyanov**
Team member 2	**Chia-Cheng Chang**
Team member 3	**Azamat Zhaksylykov**

Use the box below to explain any attempts to reach out to a non-contributing member. Type (N/A) if all members contributed.
Note: You may be required to provide proof of your outreach to non-contributing members upon request.

List

- Demonstration
  Here's a demonstration of how to work with a Python list, including creating, subsetting, modifying, and performing various operations.

Creating a list:

```
# Creating a list with integer elements
my_list = [1, 2, 3, 4, 5]

# Creating a list with mixed data types
mixed_list = [1, 'two', 3.0, (4, 5), [6, 7]]
```

Subsetting a list (accessing elements and slicing):

```
# Accessing elements by index
first_element = my_list[0]  # 1
last_element = my_list[-1]  # 5

# Slicing a list (start:stop:step)
first_three_elements = my_list[:3]  # [1, 2, 3]
every_second_element = my_list[::2]  # [1, 3, 5]
```

Modifying a list:

```
# Updating an element by index
my_list[0] = 10  # [10, 2, 3, 4, 5]

# Adding elements
my_list.append(6)  # [10, 2, 3, 4, 5, 6]
my_list.extend([7, 8, 9])  # [10, 2, 3, 4, 5, 6, 7, 8, 9]
my_list.insert(1, 1)  # [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Removing elements
my_list.remove(10)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
popped_element = my_list.pop(1)  # Removes and returns the element at index 1, i.e., 2
```

- Advantages and Disadvantages

  Advantages:

  1) Easy to use: Python lists are simple to implement and easy to understand, making them an excellent choice for beginners and experienced programmers alike.

  2) Dynamic sizing: Python lists can grow or shrink in size as needed, allowing developers to efficiently manage memory usage.

  3) Mutable: Lists are mutable, meaning you can modify their elements in-place without creating a new list. This feature makes it easy to update and manipulate data.

  4) Heterogeneous: Lists can store items of different data types, providing flexibility in storing complex data structures.

  Disadvantages:

  ➔ Slower than other data structures: Lists are not the most efficient data structure for certain operations. For example, searching for an element in a list takes O(n) time, whereas searching in a set or dictionary takes O(1) time on average.

  ➔ Memory overhead: Due to their dynamic nature and the extra space needed for pointers, Python lists have a higher memory overhead compared to other data structures like arrays or tuples.

  ➔ Inefficient for large datasets: For large datasets, lists can be slow and consume significant memory.

  ➔ Not suitable for certain applications: Lists may not be the best choice for specific use cases, such as when you need a fixed-size data structure or when you require a data structure optimized for a particular operation (e.g., using a queue or a stack).

- Python List: Best Practices and Tips - Study Aid
  - ➔ List Creation and Initialization
    Use list comprehensions for concise squares = [x**2 for x in range(1, 11)]
    e, readable code:
    squares = [x**2 for x in range(1, 11)]

  - ➔ Initialize lists of a specific size using a default value:
    default_list = [0] * 10 # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

  - ➔ Accessing and Modifying Elements
    - ◆ Use negative indices to access elements from the end of the list:
      last_element = my_list[-1]

    - ◆ Prefer slice assignments for readability and performance when modifying multiple elements:
      my_list[1:4] = [10, 20, 30]

  - ➔ List Comprehensions
    - ◆ Use list comprehensions for filtering and mapping operations:
      even_numbers = [x for x in my_list if x % 2 == 0]
      - ➔ Avoid complex or nested list comprehensions, as they can be difficult to read.

  - ➔ Memory and Performance Considerations
    - ◆ Be cautious when using the * operator with mutable objects (e.g., lists of lists), as it creates shallow copies that reference the same object.
    - ◆ Use built-in functions like enumerate() for looping over elements and their indices:
      for idx, val in enumerate(my_list):
        print(idx, val)

  - ➔ Miscellaneous Tips
    - ◆ Use the built-in sorted() function to create a sorted copy of a list without modifying the original list.
    - ◆ Use in and not in for membership testing instead of list.index() to avoid raising an exception when the element is not found.
    - ◆ Remember that lists are mutable, so be cautious when passing them as function arguments to avoid unintended side effects.

Dictionaries:

- Demonstration
  Here's a demonstration of how to work with a Python dictionary, including creating, subsetting, modifying, and performing various operations.

Creating a dictionary:

```
# Creating a dictionary. You need key-value pairs to do it and curly brackets are used to define the dictionary.
my_dict = { “key1”:“value1”, “key2”: “value2”, “key3”: “value3” }
```

Subsetting a dictionary (accessing values and keys):

```
# Accessing values using keys
value1 = my_dict[“key1”]  # “value1”

# Getting all keys or values
values =  my_dict.values() #  dict_values([“value1”, “value2”, “value3”])
keys =  my_dict.keys() #  dict_keys([“key1”, “key2”, “key3”])
```

Modifying a dictionary:

```
# Updating a value by key
my_dict[“key1”] = “new_value1”  # { “key1”:“new_value1”, “key2”: “value2”,“key3”: “value3” }

# Adding key-value pair
my_dict[“key4”]= “value4” # { “key1”:“new_value1”, “key2”: “value2”,“key3”: “value3”,“key4”: “value4” }

# Removing key-value pair
del my_dict[“key1”] #  { “key2”: “value2”,“key3”: “value3”,“key4”: “value4” }
my_dict.pop(“key4”) #  {“key2”: “value2”,“key3”: “value3”}
```

- Advantages and Disadvantages

Advantages:

- ➔ Easy to use: Python dictionaries are simple to implement and easy to understand, making them an excellent choice for beginners and experienced programmers alike.
- ➔ Dynamic sizing: Python lists can grow or shrink in size as needed, allowing developers to efficiently manage memory usage.
- ➔ Mutable: Dictionaries are mutable, meaning you can modify their elements in-place without creating a new dictionary. This feature makes it easy to update and manipulate data.
- ➔ Heterogeneous: Dictionaries can store items of different data types, providing flexibility in storing complex data structures.
- ➔ Fast lookups: Dictionaries use hash tables and efficiency of algorithms for lookups is O(1).

Disadvantages:

- ➔ Unordered: Dictionaries do not maintain order when key-value pairs are added. Therefore, it is not a suitable data structure for problems where order is important.
- ➔ Memory overhead: Due to their dynamic nature and the extra space needed for pointers, Python dictionaries have a higher memory overhead compared to other data structures like arrays or tuples.
- ➔ Key restrictions: Keys of the dictionary are unique and immutable. The main reason for keys to be unique and immutable is to avoid collusion problems of dictionaries.

- Python Dictionary: Best Practices and Tips - Study Aid
  - ➔ Dictionary Creation and Initalization
    Use dictionary comprehensions for concise and readable code:
    my_dict = {k: v for k, v in zip(keys_list, values_list)}, where zip() is a python function returning zip object.

  - ➔ Initialize dictionaries with default values using defaultdict():
    from collections import defaultdict
    my_dict=defaultdict(int)

  - ➔ Accessing and Modifying Elements
    - ◆ Use the get() method to access dictionary values
      value = my_dict.get(key)

    - ◆ Use setdefault() method to modify or add key-value pairs.
      my_dict.setdefault(key,value)

  - ➔ Dictionary Comprehensions
    - ◆ Use dictionary comprehensions for filtering and mapping operations:
      even_dict = {k:v for k,v in my_dict.items() if v % 2 == 0}

  - ➔ Memory and Performance Considerations
    - ◆ Be cautious when using dictionaries with mutable keys, as the keys must be immutable.
    - ◆ Use built-in functions like keys(), values(), and items() for iterating over dictionary:
      for v in my_dict.values():
      print(value)

  - ➔ Miscellaneous Tips
    - ◆ For membership testing use in and not in operators.
    - ◆ Remember that dictionaries are mutable, so be cautious when passing them as function arguments to avoid unintended side effects.

Numpy Arrays:

Numpy is an object-oriented python library that uses a homogeneous multidimensional array. With simple words said this means that it is a table (mostly numerical, but not always) with the same type of data that is indexed by a tuple and the indices can NOT be negative numbers.

The more important attributes of a ndarray object are [1]:

- **ndarray.ndim**
- **ndarray.shape**
- **ndarray.size**
- **ndarray.dtype**
- **ndarray.itemsize**
- **ndarray.data**

How to initiate a numpy.array:

```
import numpy as np
>>>  example_array_a = np.array([2, 3, 4])
>>> example_array_a
array([2, 3, 4])
```

```
example_array_b = = np.array([(1.5, 2, 3), (4, 5, 6)])
>>> example_array_b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

You can even explicitly specify the type of the array on the exact creation like this:

```
example_array_c = np.array([[1, 2], [3, 4]], dtype=complex)
```

```
>>> example_array_c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

As you seen it can work even with complex numbers.

Also with nynpy array as said previously, you can create n-dimentional arrays where:

- one dimensional array is a vector.
- 2-dimensional array is a 2d matrix
- 3 and more dimensional arrays are n-dimensional matrices.

Slicing and indexing:

```
example_array_data = np.array([1, 2, 3])

>>> example_array_data [1]
2
>>> example_array_data [0:2]
array([1, 2])
>>> example_array_data [1:]
array([2, 3])
>>> example_array_data [-2:]
array([2, 3])
```

You can split this array like this:

```
example_array_x = np.arange(1, 51).reshape(5, 10)

>>> example_array_x

array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15, 16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25, 26, 27, 28, 29, 30],
       [31, 32, 33, 34, 35, 36, 37, 38, 39, 40],
       [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]])
```

Or like this:

```
np.hsplit(example_array_x, 5)

[array([[ 1,  2],
        [11, 12],
        [21, 22],
        [31, 32],
        [41, 42]]),
 array([[ 3,  4],
        [13, 14],
        [23, 24],
        [33, 34],
        [43, 44]]),
 array([[ 5,  6],
        [15, 16],
        [25, 26],
        [35, 36],
        [45, 46]]),
 array([[ 7,  8],
        [17, 18],
        [27, 28],
        [37, 38],
        [47, 48]]),
 array([[ 9, 10],
```

[19, 20],
[29, 30],
[39, 40],
[49, 50]])]

Mathematical operation with arrays:

```
a = np.array([1, 2, 3, 4])

>>> a.sum()
10
```

Multiplying array with a number:

```
example_array_data = np.array([1.0, 2.0])
>>> example_array_data * 1.6
array([1.6, 3.2])
```

Other useful operations:

```
example_array_data .max()
2.0
>>> example_array_data .min()
1.0
```

Advantages:

1. **Efficiency**: NumPy arrays are designed to handle large data sets and can perform mathematical operations much faster than Python's built-in list data structure. They are optimized thanks to the C++ programing language and thus are working really fast

2. **Flexibility**: They can work with data of different types and dimensions, making it useful when working with complex data sets. Also, NumPy arrays are a useful tool for scientific computing, because they provide a big range of mathematical functions and operations to apply over arrays (vectors and matrices).

3. **Convenience**: NumPy has prebuild functions for working especially with arrays, that make them very fast and useful. This is how with a few lines of code you can reshape, stack and split arrays which can become handy when working with big and complex datasets.

Disadvantages:

1. **Learning curve:** NumPy arrays can be more difficult to work with, especially if you are not familiar with linear algebra.

2. **Less flexibility when adding/removing elements:** Compared to lists (which are mutable) arrays are immutable, so they have a fixed size and shape. So when you want to add or remove an element you have to create an entirely new array and this as you can imagine is not as convenient as simply appending or removing elements like you can do when using lists.

3. **Memory usage:** Since arrays have additional metadata like size, shape, and data type. This information must be saved somewhere which takes up additional memory. Nowadays computers have enough memory but imagine if you are working with microcontrollers that have limited memory. This can become an issue.

REFERENCES:

1. NumPy quickstart — NumPy v1.25.dev0 Manual
2. Introduction to NumPy (w3schools.com)
3. NumPy Illustrated: The Visual Guide to NumPy | by Lev Maximov | Better Programming
4. **https://res.cloudinary.com/dyd911kmh/image/upload/v1676302459/Marketing/Blog/Numpy_Cheat_Sheet.pdf**
5. **https://intellipaat.com/blog/tutorial/python-tutorial/numpy-cheat-sheet/**
6. **https://web.itu.edu.tr/iguzel/files/Python_Cheat_Sheets.pdf**
7. **https://docs.python.org/3/tutorial/datastructures.html**
8. **https://www.w3schools.com/python/python_dictionaries.asp**
9. **https://www.w3schools.com/python/python_dictionaries_access.asp**
10. **https://www.w3schools.com/python/python_dictionaries_change.asp**
11. **https://www.w3schools.com/python/python_dictionaries_add.asp**
12. **https://www.w3schools.com/python/python_dictionaries_remove.asp**
13. **https://www.w3schools.com/python/python_dictionaries_loop.asp**