Object Oriented Programming Using Python

Teemu Matilainen

teemu.matilainen@savonia.fi

Lecture 1#

Sequence Types: lists, tuples

Mapping Type: dictionaries

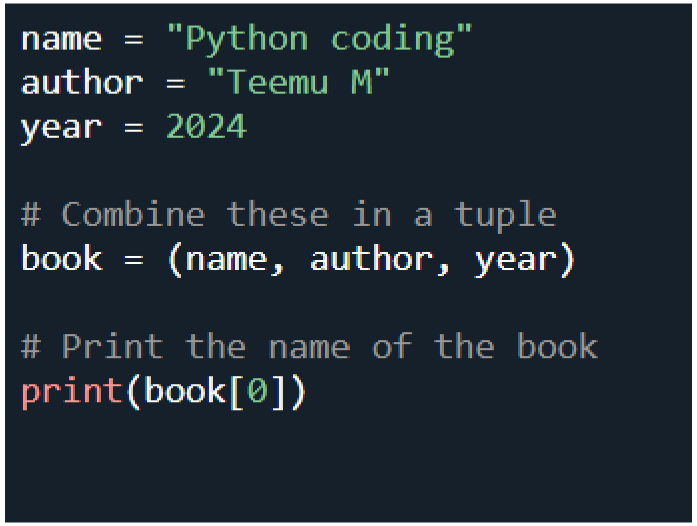
Text type: strings,…

etc.

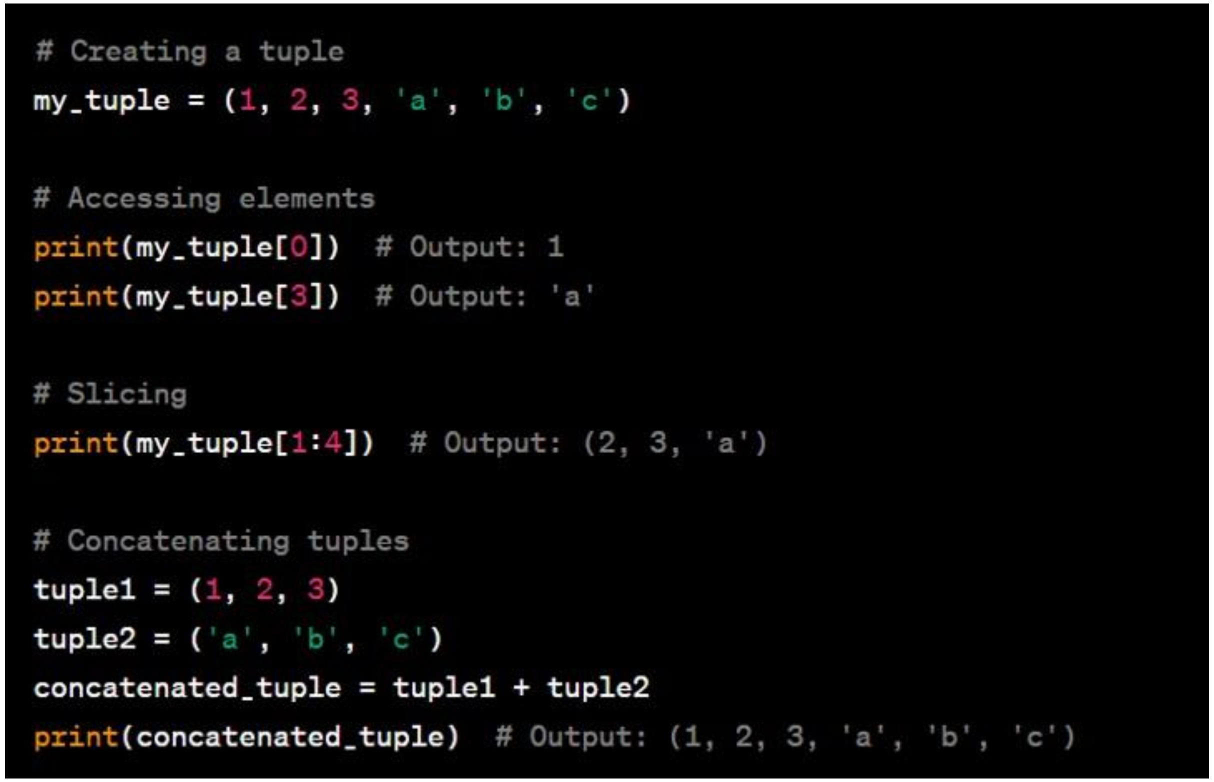
Tuple:

In Python, a tuple is an ordered, immutable collection of elements. Tuples are similar to lists, but the main difference is that tuples cannot be modified once they are created. They are defined using parentheses **()**.

Here's a couple of basic examples of creating a tuple:



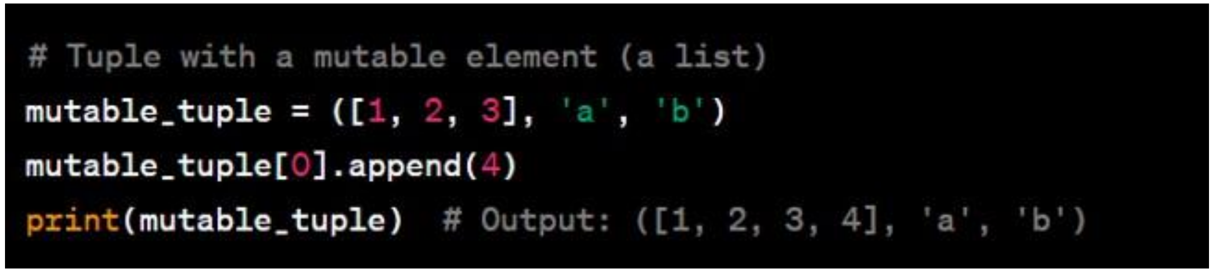
```python
name = "Python coding"
author = "Teemu M"
year = 2024

# Combine these in a tuple
book = (name, author, year)

# Print the name of the book
print(book[0])
```



```
# Creating a tuple
my_tuple = (1, 2, 3, 'a', 'b', 'c')

# Accessing elements
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: 'a'

# Slicing
print(my_tuple[1:4])  # Output: (2, 3, 'a')

# Concatenating tuples
tuple1 = (1, 2, 3)
tuple2 = ('a', 'b', 'c')
concatenated_tuple = tuple1 + tuple2
print(concatenated_tuple)  # Output: (1, 2, 3, 'a', 'b', 'c')
```

Tuples are often used when the order and the immutability of elements are important. They can be used as keys in dictionaries (unlike lists) and are generally used in situations where the data should not be modified after creation.

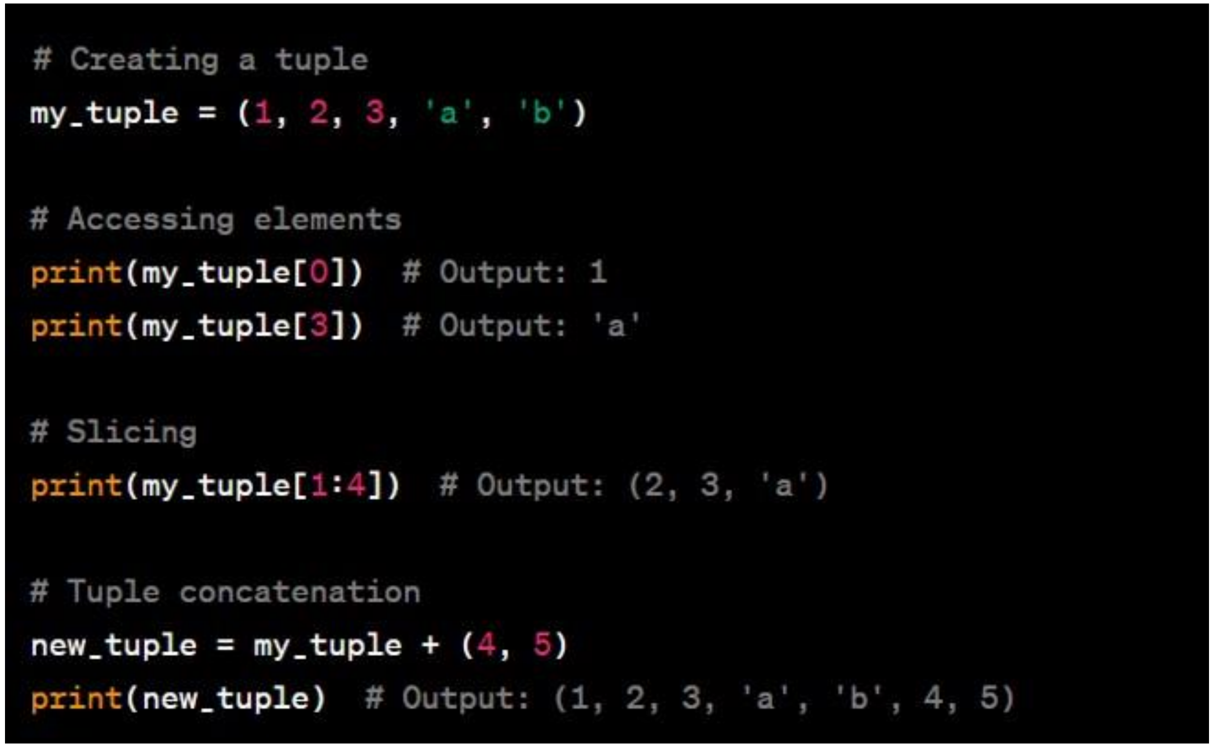
It's important to note that while the elements of a tuple cannot be changed, if the elements themselves are mutable (for example, a list), their internal state can be modified. However, the tuple still cannot be reassigned or modified directly.



```python
# Tuple with a mutable element (a list)
mutable_tuple = ([1, 2, 3], 'a', 'b')
mutable_tuple[0].append(4)
print(mutable_tuple)  # Output: ([1, 2, 3, 4], 'a', 'b')
```

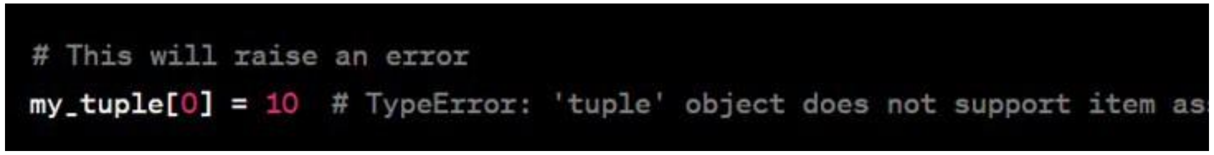
In the example above, the list inside the tuple is mutable, so its content can be modified, but the tuple itself cannot be changed or reassigned.

Another example:



```python
# Creating a tuple
my_tuple = (1, 2, 3, 'a', 'b')

# Accessing elements
print(my_tuple[0])  # Output: 1
print(my_tuple[3])  # Output: 'a'

# Slicing
print(my_tuple[1:4])  # Output: (2, 3, 'a')

# Tuple concatenation
new_tuple = my_tuple + (4, 5)
print(new_tuple)  # Output: (1, 2, 3, 'a', 'b', 4, 5)
```

Remember, since tuples are immutable, you can't do operations that modify the tuple in-place like you can with lists.



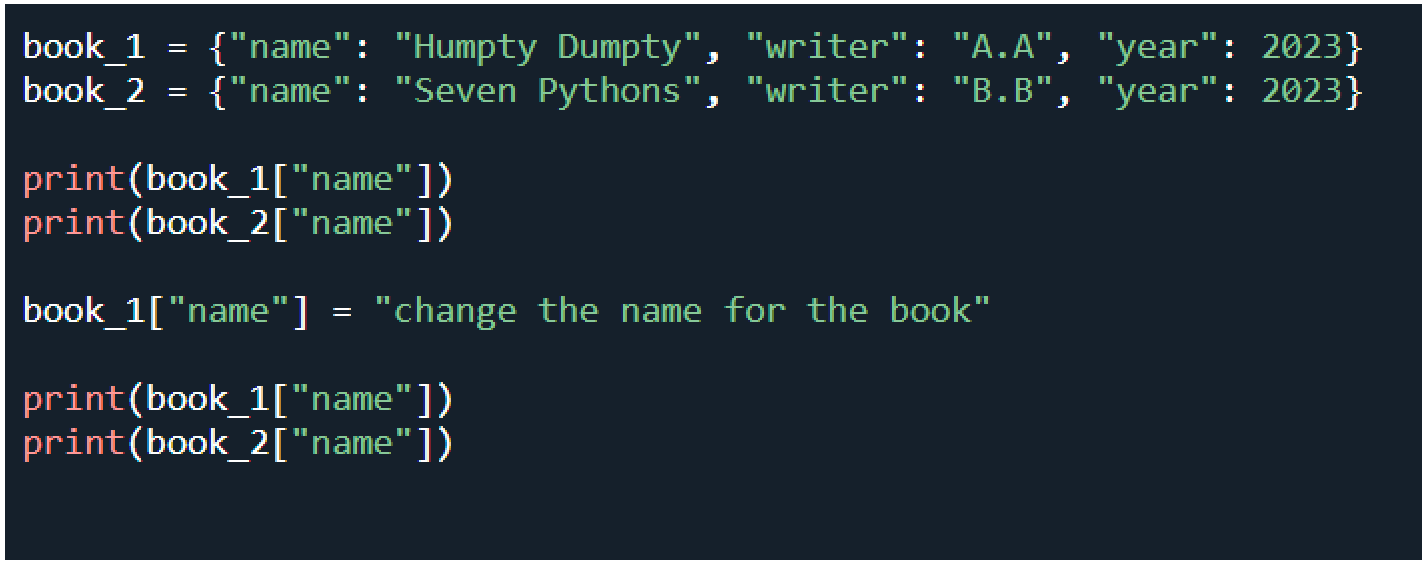
```python
# This will raise an error
my_tuple[0] = 10  # TypeError: 'tuple' object does not support item as
```

Tuples are useful when you want to create a collection of items that should not be changed throughout the program. They are also often used for functions that return multiple values.

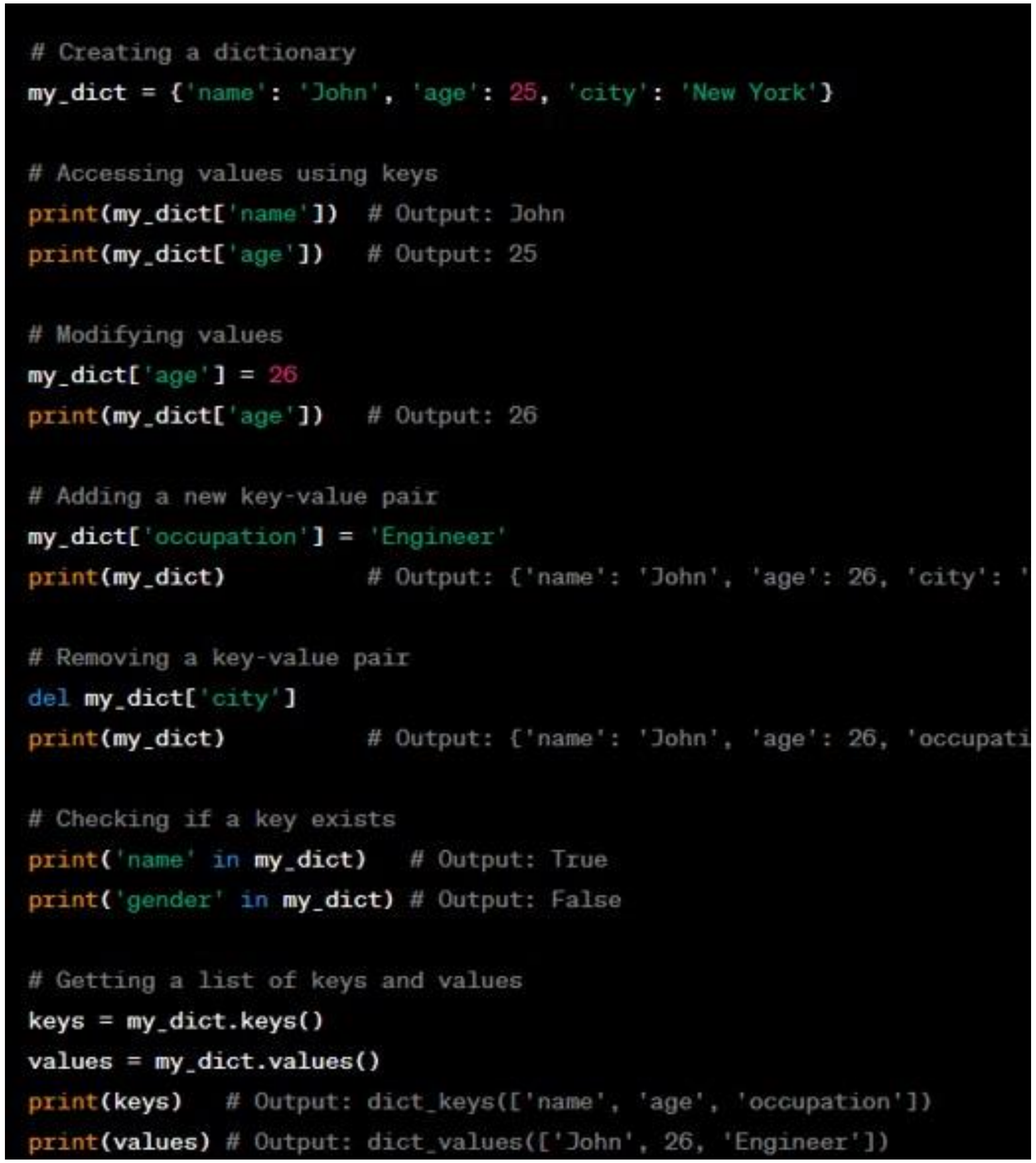
Dictionary:

Dictionary is a built-in data type that allows you to store and retrieve data in key-value pairs. Dictionaries are defined using curly braces **{}** and consist of a set of key-value pairs, where each key must be unique.

Here's a basic example of creating and using a dictionary:



```python
book_1 = {"name": "Humpty Dumpty", "writer": "A.A", "year": 2023}
book_2 = {"name": "Seven Pythons", "writer": "B.B", "year": 2023}

print(book_1["name"])
print(book_2["name"])

book_1["name"] = "change the name for the book"

print(book_1["name"])
print(book_2["name"])
```



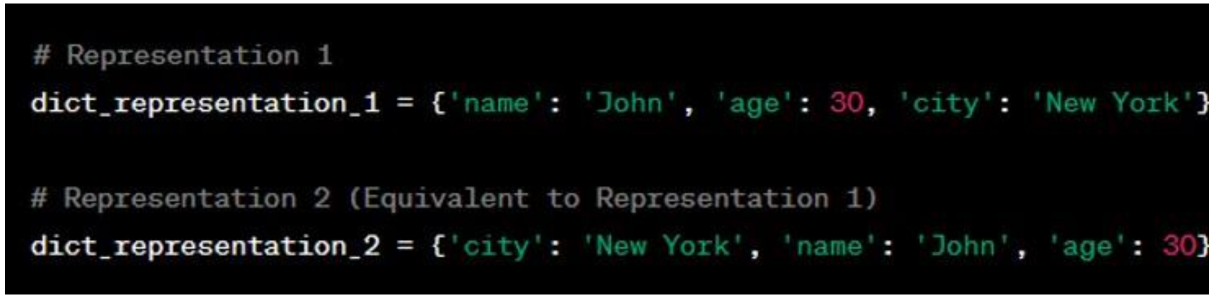
```python
# Creating a dictionary
my_dict = {'name': 'John', 'age': 25, 'city': 'New York'}

# Accessing values using keys
print(my_dict['name'])  # Output: John
print(my_dict['age'])   # Output: 25

# Modifying values
my_dict['age'] = 26
print(my_dict['age'])   # Output: 26

# Adding a new key-value pair
my_dict['occupation'] = 'Engineer'
print(my_dict)          # Output: {'name': 'John', 'age': 26, 'city': '

# Removing a key-value pair
del my_dict['city']
print(my_dict)          # Output: {'name': 'John', 'age': 26, 'occupati

# Checking if a key exists
print('name' in my_dict)   # Output: True
print('gender' in my_dict) # Output: False

# Getting a list of keys and values
keys = my_dict.keys()
values = my_dict.values()
print(keys)   # Output: dict_keys(['name', 'age', 'occupation'])
print(values) # Output: dict_values(['John', 26, 'Engineer'])
```

Dictionaries are widely used in Python for tasks such as storing configuration settings, representing JSON-like data structures, and more. They provide a flexible and efficient way to manage and organize data.

It's important to note that starting from Python 3.7, dictionaries maintain the order of insertion of items. This means that the order in which key-value pairs are added to the dictionary is preserved when iterating over the dictionary. Prior to Python 3.7, dictionaries did not guarantee any specific order.

In Python, dictionaries can have different structurally identical representations. Here are two ways to represent the same dictionary:

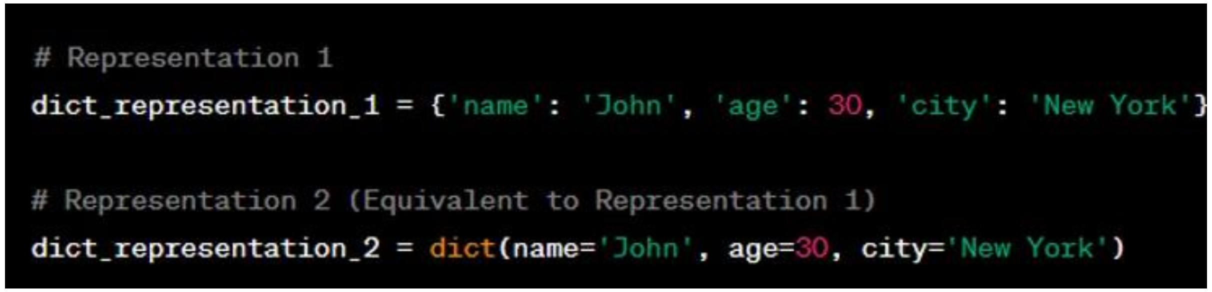
Using Literal Notation:



```
# Representation 1
dict_representation_1 = {'name': 'John', 'age': 30, 'city': 'New York'}

# Representation 2 (Equivalent to Representation 1)
dict_representation_2 = {'city': 'New York', 'name': 'John', 'age': 30}
```

In the above example, **dict_representation_1** and **dict_representation_2** are structurally identical even though the order of key-value pairs is different. Starting from Python 3.7, dictionaries preserve the order of insertion, so the order of key-value pairs is maintained when iterating over the dictionary. However, in any version of Python, these two dictionaries are considered equivalent.

Using the dict() Constructor:



```python
# Representation 1
dict_representation_1 = {'name': 'John', 'age': 30, 'city': 'New York'}

# Representation 2 (Equivalent to Representation 1)
dict_representation_2 = dict(name='John', age=30, city='New York')
```

In this example, **dict_representation_1** and **dict_representation_2** are again structurally identical, even though the second representation uses the **dict()** constructor with keyword arguments. Both representations create dictionaries with the same key-value pairs.

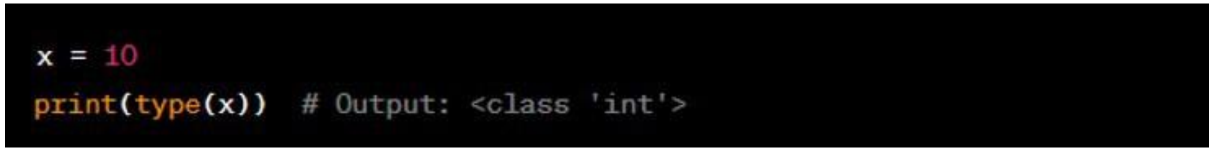
It's important to note that when comparing dictionaries for equality, Python considers them equal if they have the same set of key-value pairs, regardless of the order of insertion or the method used to create them.

Python Objects:

In Python, everything is an object. This means that every value in Python is an instance of an object, and every object has a type (class). Whether it's a simple data type like integers and strings or more complex structures like lists and dictionaries, everything is an object.

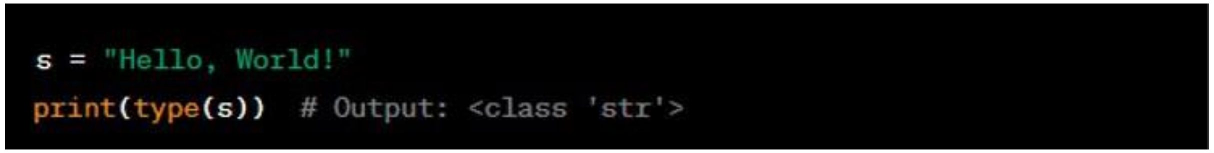
Here are some examples of Python objects:

1. **Integers:**



```python
x = 10
print(type(x))  # Output: <class 'int'>
```

2. **Strings:**



```python
s = "Hello, World!"
print(type(s))  # Output: <class 'str'>
```

3. **Lists:**

```python
my_list = [1, 2, 3, 4]
print(type(my_list))  # Output: <class 'list'>
```

4. Dictionaries:

```python
my_dict = {'key': 'value', 'name': 'John'}
print(type(my_dict))  # Output: <class 'dict'>
```

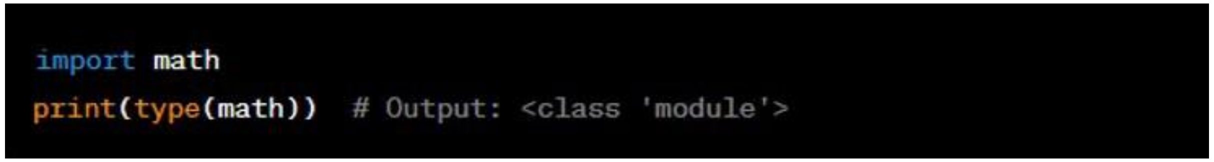
5. Functions:

```python
def my_function():
    print("Hello from a function!")

print(type(my_function))  # Output: <class 'function'>
```

6. Classes:

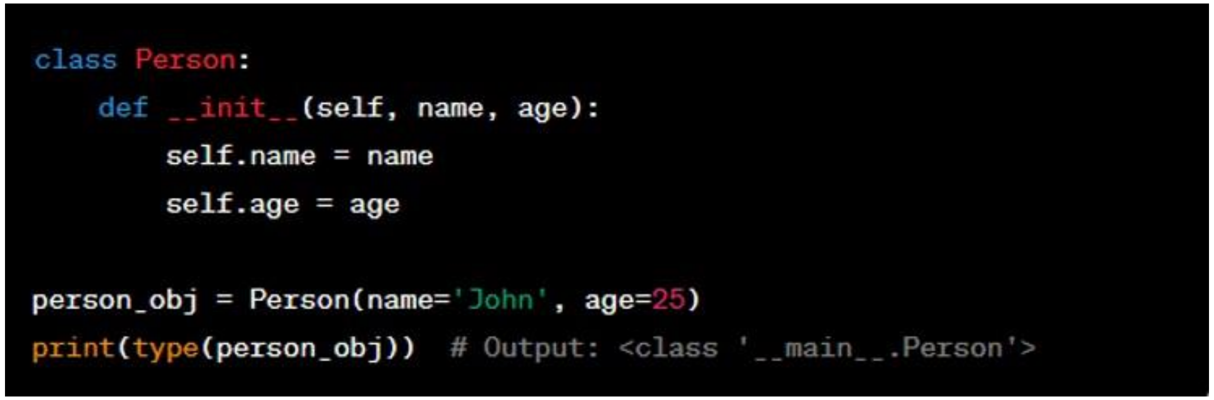
```python
class MyClass:
    pass

obj = MyClass()
print(type(obj))  # Output: <class '__main__.MyClass'>
```

7. **Modules:** Modules themselves are objects.



```python
import math
print(type(math))  # Output: <class 'module'>
```

8. **Custom Objects:** You can create your own classes and instantiate objects from them.



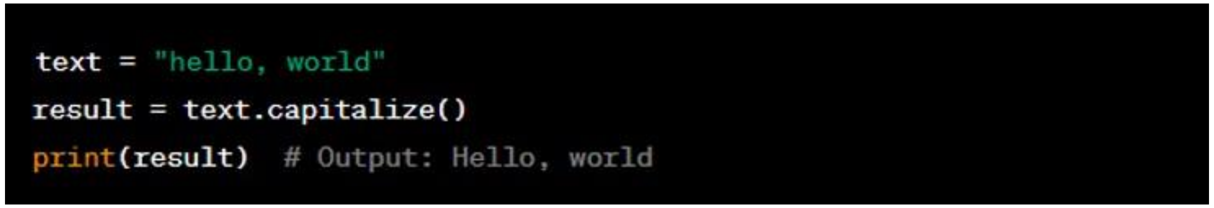
```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person_obj = Person(name='John', age=25)
print(type(person_obj))  # Output: <class '__main__.Person'>
```

Understanding that everything in Python is an object is fundamental to the language. It means that objects can be passed around, assigned to variables, and manipulated in various ways, providing a flexible and consistent programming model.

Objects and methods:

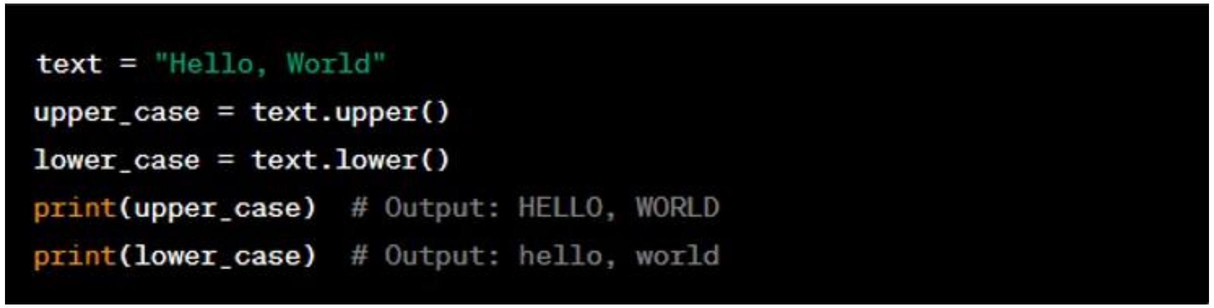
In Python, strings are objects, and they come with a variety of built-in methods that you can use to manipulate and work with strings. Here are some commonly used string methods:

1. **str.capitalize():** Returns a copy of the string with its first character capitalized.



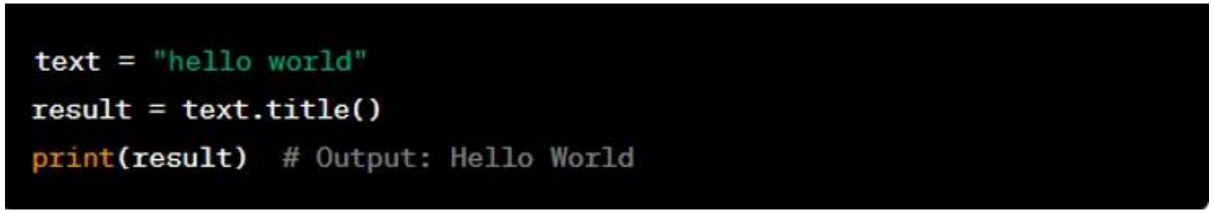
```python
text = "hello, world"
result = text.capitalize()
print(result)  # Output: Hello, world
```

2. **str.upper() and str.lower()**: Returns a copy of the string with all characters in uppercase or lowercase.



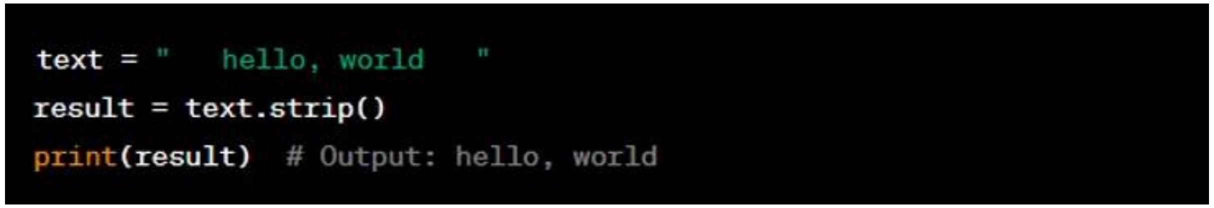
```python
text = "Hello, World"
upper_case = text.upper()
lower_case = text.lower()
print(upper_case)  # Output: HELLO, WORLD
print(lower_case)  # Output: hello, world
```

3. **str.title()**: Returns a titlecased version of the string, where words start with uppercase letters and the remaining characters are lowercase.



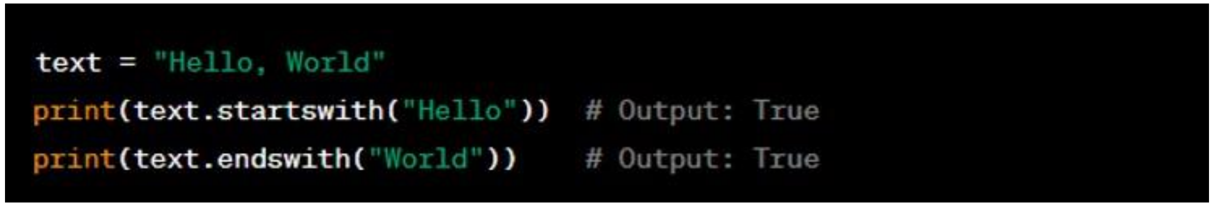
```python
text = "hello world"
result = text.title()
print(result)  # Output: Hello World
```

4. **str.strip()**: Returns a copy of the string with leading and trailing whitespace removed.



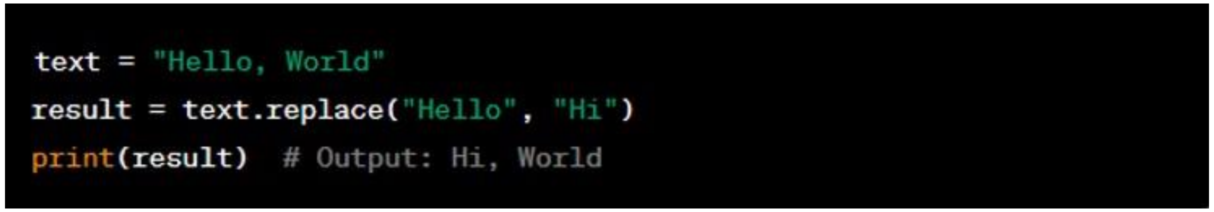
```python
text = "   hello, world   "
result = text.strip()
print(result)  # Output: hello, world
```

5. **str.startswith(prefix) and str.endswith(suffix)**: Returns **True** if the string starts with the specified prefix or ends with the specified suffix; otherwise, it returns **False**.



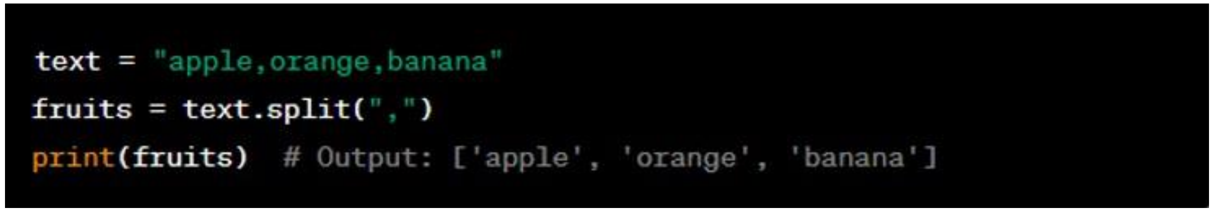
```
text = "Hello, World"
print(text.startswith("Hello"))  # Output: True
print(text.endswith("World"))    # Output: True
```

6. **str.replace(old, new)**: Returns a copy of the string with all occurrences of the substring **old** replaced by **new**.



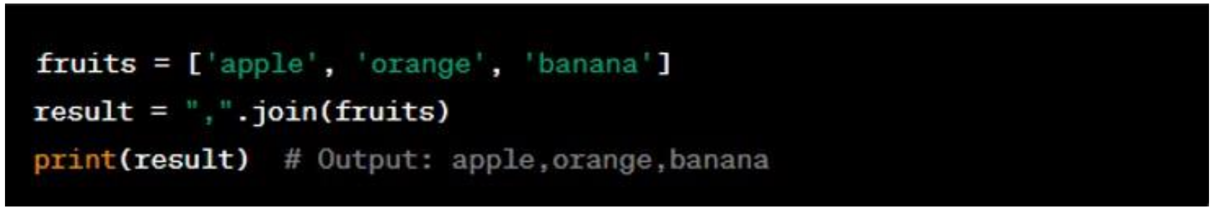
```
text = "Hello, World"
result = text.replace("Hello", "Hi")
print(result)  # Output: Hi, World
```

7. **str.split(separator)**: Splits the string into a list of substrings based on the specified **separator**.



```
text = "apple,orange,banana"
fruits = text.split(",")
print(fruits)  # Output: ['apple', 'orange', 'banana']
```

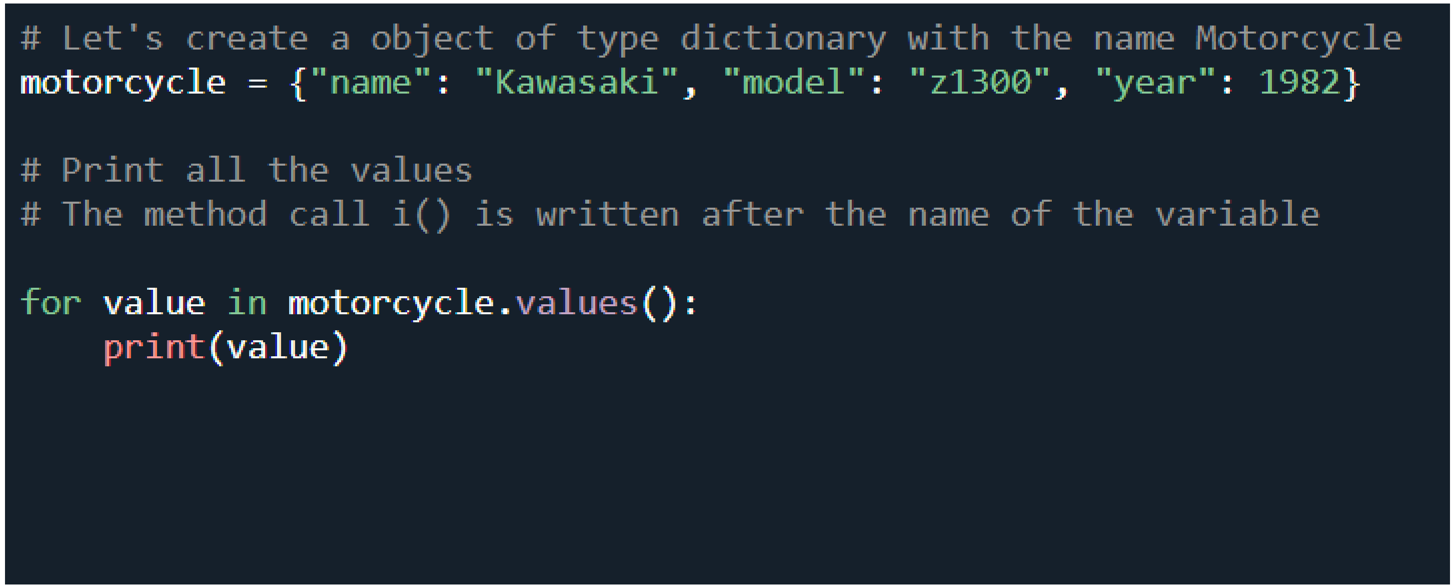
8. **str.join(iterable)**: Joins the elements of an iterable (e.g., a list) into a single string using the original string as a separator.



```python
fruits = ['apple', 'orange', 'banana']
result = ",".join(fruits)
print(result)  # Output: apple,orange,banana
```

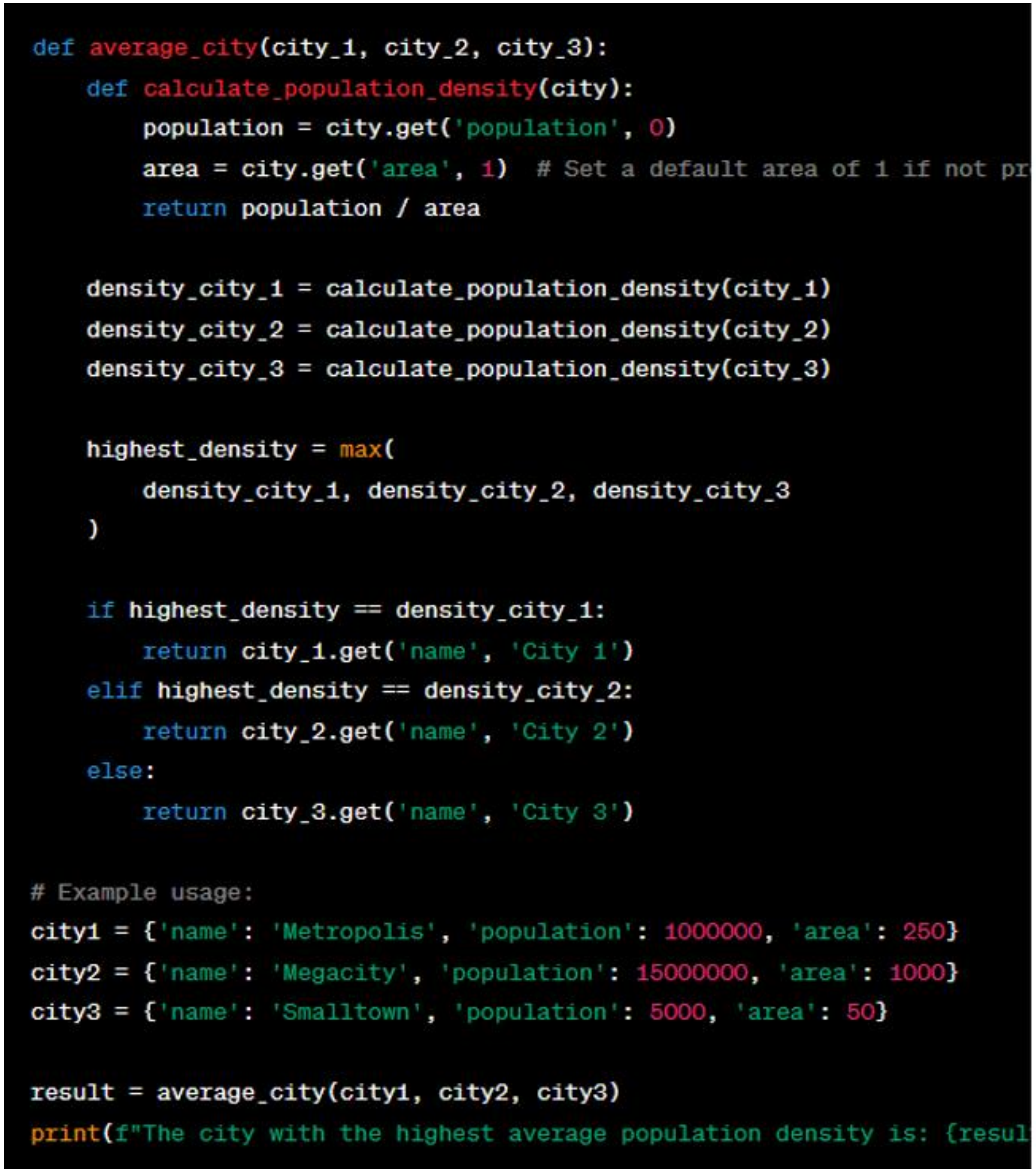
These are just a few examples of the many string methods available in Python. String methods provide powerful tools for string manipulation and are essential for working with text data in Python.

Here’s an example for usage of dictionary object and it’s method **values**:



```
# Let's create a object of type dictionary with the name Motorcycle
motorcycle = {"name": "Kawasaki", "model": "z1300", "year": 1982}

# Print all the values
# The method call i() is written after the name of the variable

for value in motorcycle.values():
    print(value)
```

Here's an example of a function named **average_city** that takes three dictionary objects representing cities and calculates and returns the city with the highest average population density. Each city dictionary is expected to have a 'name' key and a 'population' key.



```
def average_city(city_1, city_2, city_3):
    def calculate_population_density(city):
        population = city.get('population', 0)
        area = city.get('area', 1)  # Set a default area of 1 if not pr
        return population / area

    density_city_1 = calculate_population_density(city_1)
    density_city_2 = calculate_population_density(city_2)
    density_city_3 = calculate_population_density(city_3)

    highest_density = max(
        density_city_1, density_city_2, density_city_3
    )

    if highest_density == density_city_1:
        return city_1.get('name', 'City 1')
    elif highest_density == density_city_2:
        return city_2.get('name', 'City 2')
    else:
        return city_3.get('name', 'City 3')

# Example usage:
city1 = {'name': 'Metropolis', 'population': 1000000, 'area': 250}
city2 = {'name': 'Megacity', 'population': 15000000, 'area': 1000}
city3 = {'name': 'Smalltown', 'population': 5000, 'area': 50}

result = average_city(city1, city2, city3)
print(f"The city with the highest average population density is: {resul
```

In this example, each city is represented by a dictionary with a 'name' key, a 'population' key, and an optional 'area' key (defaulted to 1 if not provided). The calculate_population_density function calculates the population density for a city, and the average_city function compares the population densities of the three cities and returns the name of the city with the highest average population density.

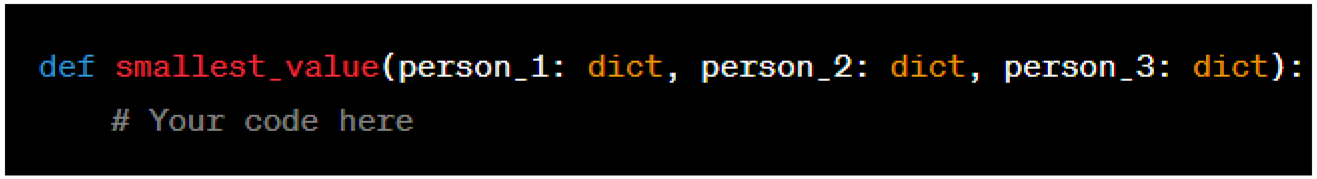
Assignment 1#: Find the Smallest Value

Objective:

- Write a Python function named **smallest_value** that takes three dictionary objects as its arguments.

Instructions:

1. **Function Declaration:**
    - Declare a function named **smallest_value** with the following signature:



```python
def smallest_value(person_1: dict, person_2: dict, person_3: dict):
    # Your code here
```

2. **Dictionary Structure:**
    - Each dictionary (**person_1**, **person_2**, **person_3**) represents a person's data and contains keys for different attributes (e.g., 'age', 'height', 'weight'). Assume each dictionary has the same set of keys.

3. **Find the Smallest Value:**
    - In the function, compare the values of a specific attribute (e.g., 'age') from all three dictionaries.
    - Identify the person with the smallest value for that attribute.
4. **Return Result:**
    - Return a tuple containing the person's name and the corresponding attribute value.
5. **Example:**
    - For example, calling **smallest_value({'name': 'Alice', 'age': 25, 'height': 160}, {'name': 'Bob', 'age': 30, 'height': 175}, {'name': 'Charlie', 'age': 22, 'height': 155})** might return **('Charlie', 22)** if we are comparing the 'age' attribute.

Assignment 2#

Write a function named calculate_row_sums(my_m_matrix: list), where you give an integer matrix as its argument.

The function adds a new element on each row in the matrix. This element contains the sum of the other elements on that row.

```
# Example:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]]
```

Result:

```
[1, 2, 3, 6]
[4, 5, 6, 15]
[7, 8, 9, 24]
```