Extra practice – W6S1

The tuple type

The tuple type

- **Tuples** are **collections of variables**, very similar to **lists**, but have their elements listed between **parentheses ()**, instead of brackets **[]**.

```
1  # Lists vs tuples
2  my_list = [1, 3, 5, 7, 9]
3  my_tuple = (1, 3, 5, 7, 9)
4  print(my_list)
5  print(my_tuple)
6  print(type(my_list))
7  print(type(my_tuple))
```

```
[1, 3, 5, 7, 9]
(1, 3, 5, 7, 9)
<class 'list'>
<class 'tuple'>
```

The tuple type

- **Tuples** are **collections of variables**, very similar to **lists**, but have their elements listed between **parentheses ()**, instead of brackets **[]**.

- They share many functions with lists, such as the **len()** function for instance.

```
1  # Lists vs tuples
2  my_list = [1, 3, 5, 7, 9]
3  my_tuple = (1, 3, 5, 7, 9)
4  print(my_list)
5  print(my_tuple)
6  print(type(my_list))
7  print(type(my_tuple))
```

```
[1, 3, 5, 7, 9]
(1, 3, 5, 7, 9)
<class 'list'>
<class 'tuple'>
```

```
1  # Most functions on lists work on tuples
2  print(len(my_list))
3  print(len(my_tuple))
```

```
5
5
```

The tuple type

As with **lists**, **tuples** objects can be

- **Indexed**,
- **Sliced**,
- **Traversed** using **for** loops.

```
1  # Indexing and slicing works on tuples
2  my_list = [1, 3, 5, 7, 9]
3  my_tuple = (1, 3, 5, 7, 9)
4  print(my_list[0])
5  print(my_tuple[0])
6  print(my_list[-1])
7  print(my_tuple[-1])
8  print(my_list[1:3])
9  print(my_tuple[1:3])
```

```
1
1
9
9
[3, 5]
(3, 5)
```

```
1  # Traversing a tuple
2  my_tuple = (1, 3, 5, 7, 9)
3  for val in my_tuple:
4      print(val)
```

```
1
3
5
7
9
```

The tuple type

As with **lists**, **tuples** objects can be

- **Indexed**,
- **Sliced**,
- **Traversed** using **for** loops.

However, the **major difference** between lists and tuples is that tuples, just like strings, are **UNMUTABLE**.

Unmutable: values can only be changed on creation. No updates.

```
1  # Tuples however are unmutable
2  # Cannot be updated as in lists
3  my_list = [1, 3, 5, 7, 9]
4  my_list[3] = "Hello"
5  print(my_list)
6  my_tuple = (1, 3, 5, 7, 9)
7  my_tuple[3] = "Hello" # Does not work!
8  print(my_tuple)
```

```
[1, 3, 5, 'Hello', 9]

---------------------------------------------------------------
TypeError                                 Traceback (most
<ipython-input-5-98dfe8c39635> in <module>
      5 print(my_list)
      6 my_tuple = (1, 3, 5, 7, 9)
----> 7 my_tuple[3] = "Hello" # Does not work!
      8 print(my_tuple)

TypeError: 'tuple' object does not support item assignment
```

The tuple type

- In addition, most **methods** from lists will **not work** on tuples.
- For instance, **append()** does not work on tuples.

```
1  # Most methods of lists do not work on tuples
2  my_list = [1, 3, 5, 7, 9]
3  my_list.append(11)
4  print(my_list)
5  my_tuple = (1, 3, 5, 7, 9)
6  my_tuple.append(11) # Does not work!
7  print(my_tuple)
```

```
[1, 3, 5, 7, 9, 11]

---------------------------------------------------------------
AttributeError                            Traceback (most
<ipython-input-6-d54ba0da70af> in <module>
      4 print(my_list)
      5 my_tuple = (1, 3, 5, 7, 9)
----> 6 my_tuple.append(11) # Does not work!
      7 print(my_tuple)

AttributeError: 'tuple' object has no attribute 'append'
```

The tuple type

- In addition, most **methods** from lists will **not work** on tuples.
- For instance, **append()** does not work on tuples.

- However, you may jump from lists to tuples types, and vice-versa, if needed.

```
1  # However, easy to convert
2  # from lists to tuples and vice versa
3  my_tuple = (1, 3, 5, 7, 9)
4  my_tuple_as_list = list(my_tuple)
5  print(my_tuple_as_list)
6  my_tuple_as_list.append(11)
7  print(my_tuple_as_list)
8  my_tuple_back = tuple(my_tuple_as_list)
9  print(my_tuple_back)
```

```
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9, 11]
(1, 3, 5, 7, 9, 11)
```

The tuple type

- **Note:** when defining functions with multiple outputs, the **type of the collection of returned values** is a **tuple**.

- When using multiple assignments, Python automatically assigns the tuple elements to each variable.

```
1  # Tuples are the default return type
2  # for functions with multiple outputs.
3  def f(x, y):
4      return x+y, x*y
```

```
1  # Tuples are the default return type
2  # for functions with multiple outputs.
3  result = f(2, 3)
4  print(result)
5  print(type(result))
6  sum_xy, mult_xy = result
7  print(sum_xy)
8  print(mult_xy)
```

```
(5, 6)
<class 'tuple'>
5
6
```

Activity 1 - Merge items in inventory

In several video games, the main character will have an inventory, i.e. a list of items that he/she is carrying at the moment. In this activity, the inventory will be defined as a list of tuples, as below.

```
inventory = [("Potion", 60), ("Iron", 45), ("Potion", 45), ("Wood", 10),
             ("Fish", 15), ("Iron", 10)]
```

Notice how each element in the list is a tuple, with two elements:

- the first element is a string, corresponding to the name of the item, currently in this item slot,
- the second element is an integer, corresponding to the number of items, currently in this item slot.

Activity 1 - Merge items in inventory

Write a function **merge_items()**, which will combine the items in the inventory. For the inventory given above, the merged inventory is:

inventory = [("Potion", 60), ("Iron", 45), ("Potion", 45), ("Wood", 10), ("Fish", 15), ("Iron", 10)]

merged_inventory = [("Potion", 105), ("Iron", 55), ("Wood", 10), ("Fish", 15)]

Matt’s Great advice #?

Matt’s Great Advice #?: Lists are more versatile, sets/tuples are more efficient.

What should we prefer between lists and tuples? Well, it depends.

- Overall, **lists** are **more versatile** (more functions and methods).
- **Tuples** are **more efficient** for basic operations, but have less methods and functions and are unmutable.

Use both types, based on your needs, and use **types conversion** if needed!