Tuples

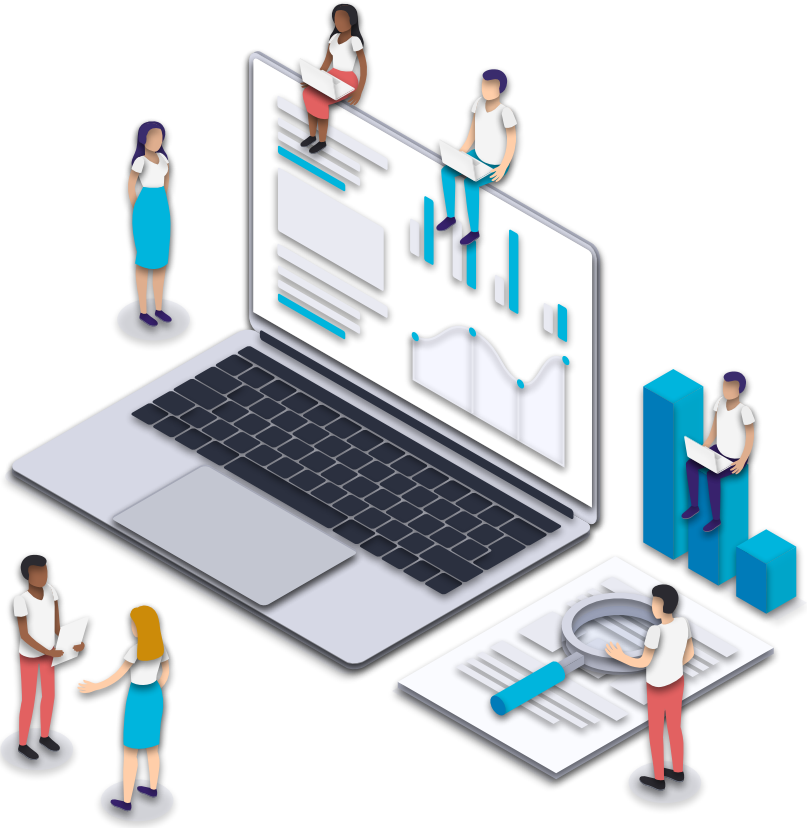


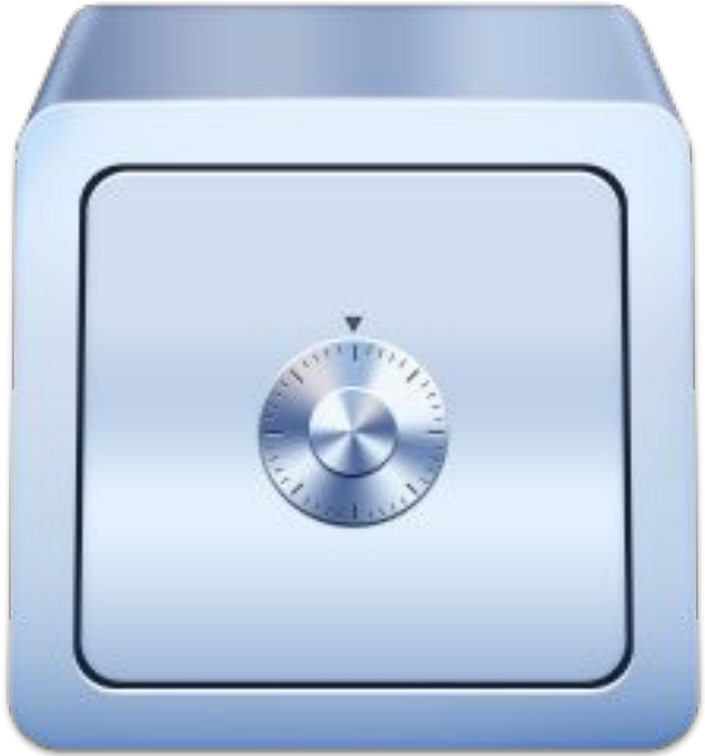
Table of Contents

- Definitions
- Creating a Tuple
- How can We Use a Tuple

Definitions

Definitions

- Your data is safe.



Definitions

- Lists vs Tuples

Creating a `tuple`

Creating a `tuple`

- We have two basic ways to create a tuple.

- `()`
- `tuple()`

Creating a `tuple`

- A `tuple` can be created by enclosing values, separated by commas, in parentheses 👉 `()`.
- Another way to create a `tuple` is to call the `tuple()` function.

- `()`
- `tuple()`

```
tuple_1 = ('h', 'a', 'p', 'p', 'y')
word = 'happy'
tuple_2 = tuple(word)
print(tuple_1)
print(tuple_2)
```

What is the output? Try to figure out in your mind...

Creating a `tuple`

- A `tuple` can be created by enclosing values, separated by commas, in parentheses 👉 `()`.
- Another way to create a `tuple` is to call the `tuple()` function.

- `()`
- `tuple()`

```
tuple_1 = ('h', 'a', 'p', 'p', 'y')
word = 'happy'
tuple_2 = tuple(word)
print(tuple_1)
print(tuple_2)
```

⚠ an iterable object can be converted into a `tuple`

```
('h', 'a', 'p', 'p', 'y')
('h', 'a', 'p', 'p', 'y')
```

Creating a `tuple` (review the pre-class)

Here is an example of creating an empty `tuple`:

```
1  empty_tuple = ()
2  print(type(empty_tuple))
3
```

Creating a `tuple` (review the pre-class)

Here is an example of creating an empty `tuple`:

```
1  empty_tuple = ()
2  print(type(empty_tuple))
3
```

```
1  <class 'tuple'>
2
```

Creating a `tuple`

- Take a look at the following example about creating a `tuple`:

```
1  my_tuple = ("Solar")
2
3  print(my_tuple, type(my_tuple), sep="\n")
4
5
6
```

What is the output? Try to figure out in your mind...

Creating a `tuple`

- Single element tuple :

```
1  my_tuple = ("Solar")
2
3  print(my_tuple, type(my_tuple), sep="\n")
4
5
6
```

Output

```
Solar
<class 'str'>
```

Creating a `tuple`

- If you want to create a single element `tuple`, an error will probably rise, unless you do not use a **comma**:

```
1  my_tuple = ("Solar",)
2
3  print(my_tuple, type(my_tuple), sep="\n")
4
5
6
```

⚠ comma makes it `tuple` type.

Output

```
('Solar',)
<class 'tuple'>
```

Creating a `tuple` (review the pre-class)

- Without parenthesis : 👇

```
1  solar = "Earth", "Venus", "Uranus"
2
3  print(solar, type(solar), sep="\n")
4
5
6
```

Creating a `tuple` (review the pre-class)

- Another way of creating a tuple :

```
1  solar = "Earth", "Venus", "Uranus"
2
3  print(solar, type(solar), sep="\n")
4
5
6
```

⚠️ Items separated with a **comma** without parentheses are accepted as `tuple` type by Python.

Output

```
('Earth', 'Venus', 'Uranus')
<class 'tuple'>
```

Creating a tuple

- `list` to `tuple`, `tuple` to `list`

```
1  my_tuple=(1, 4, 3, 4, 5, 6, 7, 4)
2
3  my_list = list(my_tuple)
4
5  print(type(my_list), my_list)
6
```

Creating a tuple

- `list` to `tuple`, `tuple` to `list`

```
1  my_tuple=(1, 4, 3, 4, 5, 6, 7, 4)
2
3  my_list = list(my_tuple)
4
5  print(type(my_list), my_list)
6
```

```
1  <class 'list'> [1, 4, 3, 4, 5, 6, 7, 4]
2
```

Creating a tuple

- `list` to `tuple`, `tuple` to `list`

```
1  my_tuple=(1, 4, 3, 4, 5, 6, 7, 4)
2
3  my_list = list(my_tuple)
4
5  print(type(my_list), my_list)
6
```

```
1  <class 'list'> [1, 4, 3, 4, 5, 6, 7, 4]
2
```

```
1  my_list = [1, 4, 3, 4, 5, 6, 7, 4]
2
3  my_tuple = tuple(my_list)
4
5  print(type(my_tuple), my_tuple)
6
```

Creating a tuple

- `list` to `tuple`, `tuple` to `list`

```
1  my_tuple=(1, 4, 3, 4, 5, 6, 7, 4)
2
3  my_list = list(my_tuple)
4
5  print(type(my_list), my_list)
6
```

```
1  <class 'list'> [1, 4, 3, 4, 5, 6, 7, 4]
2
```

```
1  my_list = [1, 4, 3, 4, 5, 6, 7, 4]
2
3  my_tuple = tuple(my_list)
4
5  print(type(my_tuple), my_tuple)
6
```

```
1  <class 'tuple'> (1, 4, 3, 4, 5, 6, 7, 4)
2
```

Creating a `tuple`

- Creating a tuple with `tuple()` function

```
1  mountain = tuple('Alps')
2  print(mountain)
3
```

Creating a `tuple`

- Creating a tuple with `tuple()` function

```
1  mountain = tuple('Alps')
2  print(mountain)
3
```

```
1  ('A', 'l', 'p', 's')
2
```

Creating a `tuple`

- Take a look at the following example :

```
tuple_1 = 'h', 'a', 'p', 'p', 'y'
tuple_2 = 1, 3, 5
print(tuple_1)
print(type(tuple_1))
print(tuple_2)
```

What is the output? Try to figure out in your mind...

Creating a `tuple`

- Considering the parentheses: As we mentioned before, There is another and not so often used way to create a `tuple`. Take a look at the following example :

```
tuple_1 = 'h', 'a', 'p', 'p', 'y'
tuple_2 = 1, 3, 5
print(tuple_1)
print(type(tuple_1))
print(tuple_2)
```

⚠ There is no parenthesis

```
('h', 'a', 'p', 'p', 'y')
<class 'tuple'>
(1, 3, 5)
```

How can We Use a `tuple`?

- (review of the pre-class)
  - Just like the **`lists`**, the **`tuples`** support indexing :

```
1  even_no = (0, 2, 4)
2  print(even_no[0])
3  print(even_no[1])
4  print(even_no[2])
5  print(even_no[3])
6
```

How can We Use a `tuple`?

- (..Continued)(review of the pre-class)
  - Just like the **`lists`**, the **`tuples`** support indexing :

```
1  even_no = (0, 2, 4)
2  print(even_no[0])
3  print(even_no[1])
4  print(even_no[2])
5  print(even_no[3])
6
```

```
1  0
2  2
3  4
4  ---------------------------------------------------------------------------
5  print(even_no[3]) : IndexError: tuple index out of range
6
```

How can We Use a `tuple`?

- (..Continued)(review of the pre-class)
  - `tuple` is **immutable**.

```
1  city_list = ['Tokyo', 'Istanbul', 'Moskow', 'Dublin']
2
3  city_tuple = tuple(city_list)
4
5  city_tuple[0] = 'New York'  # you can't assign a value
6
```

How can We Use a `tuple`?

- (..Continued)(review of the pre-class)
  - And one of the most important differences of `tuple`s from `list`s is that `tuple` object does not support item assignment. Yes, because `tuple` is immutable.

```
1  city_list = ['Tokyo', 'Istanbul', 'Moskow', 'Dublin']
2
3  city_tuple = tuple(city_list)
4
5  city_tuple[0] = 'New York'  # you can't assign a value
6
```

```
1  ---------------------------------------------------------------------------
2  TypeError: 'tuple' object does not support item assignment
3
```

Using Tuples

- **Task** :
  - Let's access, select and print the string 'six" from the following tuple. 👇

```
1  mix_tuple = ("11", 11, [2, "two", ("six", 6)], (5, "fair"))
2
```

Using Tuples

- **The code should be like this : 👇**

```
1  mix_tuple = ("11", 11, [2, "two", ("six", 6)], (5, "fair"))
2
3  str_six = mix_tuple[2][2][0]
4
5  print(str_six)
6
7
```

Using Tuples

- **Task** :
  - What is the output ? 👇

```
1  mix_tuple = ("11", 11, [2, "two", ("six", 6)], (5, "fair"))
2
3  str_six = mix_tuple[2][1:3]
4
5  print(str_six, type(str_six), sep="\n")
6
7
```

Using Tuples

- **The output :** 👇

```
1  mix_tuple = ("11", 11, [2, "two", ("six", 6)], (5, "fair"))
2
3  str_six = mix_tuple[2][1:3]
4
5  print(str_six, type(str_six), sep="\n")
6
7
```

Try to figure out how the output can be like that?

Output

```
['two', ('six', 6)]
<class 'list'>
```

Using Tuples

- **Task** :
  - Access and print the **last** item and **its type** of the following tuple using **negative indexing** method : 👇

```
1  mix_tuple = ("11", 11, [2, "two", ("six", 6)], (5, "fair"))
2
```

Using Tuples

- **The code should be like :** 👇

```
1  mix_tuple = ("11", 11, [2, "two", ("six", 6)], (5, "fair"))
2
3  last = mix_tuple[-1]
4
5  print(last, type(last), sep="\n")
6
7
```

Try to figure out how the output can be like that?

Output

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

Using Tuples

- **Task** :
  - Let's access, select and print the "fair" of the following tuple. 👇 Use **two options** which consisting of *normal* and *negative* indexing methods.

```
1  mix_tuple = ("11", 11, [2, "two", ("six", 6)], (5, "fair"))
2
```

Using Tuples

- **The code should be like :** 👇

```
1  mix_tuple = ("11", 11, [2, "two", ("six", 6)], (5, "fair"))
2
3  option_1 = mix_tuple[3][1]
4  option_2 = mix_tuple[-1][1]
5
6  print(option_1, option_2, sep = "\n")
7
8
```

Output

```
fair
fair
```

Refresh your mind with this interview question

Benefits of Immutability?

Try to write at least two things