{ Python }

Introduction

- Getting Start
- Variable and Literals
- Input and output
- Type conversion

Decision Making & Loop

- Boolean Expression
- If …else statement
- While loop
- For loop
- Break and continue
- Pass
- Control flow Example

Functions

- Function
- Variable scope and literals
- Function arguments
- Anonymous function
- Recursion

Collection and Data type

- List
- Tuples
- String
- Sets
- Dictionary

Modules and Files

- Modules
- Files
- Directory

Exceptions handling

- Exception
- Exception handling
- Custom Exception

Object - Oriented Programming

- Class and Object
- Constructors
- Inheritance
- NameSpaces

Advance Topic

- Iterators
- Generatores
- Closures
- Decorators
- Python @Property

Tuple

{ Collection Data Type }

Tuple Introduction

A tuple is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas, in a list, elements can be changed.

A tuple is created by placing all the items (elements) inside parentheses `( )`, separated by commas.

A tuple can have any number of items and they may be of different types (integer, float, list, string etc.).

```
my_tuple = ()      # empty tuple
my_tuple = (1, 2, 3) # tuple having
integers
my_tuple = (1, "Hello", 3.4) # mixed data
types
my_tuple = ('cat', [8, 4], (1, 2, 3)) #
nested tuple
```

A tuple can also be created without using parentheses. However, it's a good practice to use them.

```
my_tuple = 3, 4.6, 'dog'
print(my_tuple)  # Output: (3, 4.6, 'dog')
```

Creating Tuples with One Item

Creating a tuple with one item is a bit tricky.

Having one item within parentheses is not enough. We will need a trailing comma to indicate that it is in fact a tuple.

```
# only parentheses is not enough
t1 = ("hello")
print(type(t1))   # Output: <class 'str'>

# need a comma at the end
t2 = ("hello",)
print(type(t2))  # Output: <class 'tuple'>

# parentheses is optional
t3 = "hello",
print(type(t3))  # Output: <class 'tuple'>
```

Here, `t1` looks like a tuple but isn't. As you can see, the type of `t1` is `<class 'str'>` suggesting it's a string.

Accessing Tuple Elements

Similar to lists, we use the index operator `[]` to access tuple elements.

Suppose, a tuple has 4 items (elements) like this:

```
my_tuple = ('a', 'b', 'c', 'd')
```

We can access the first item with `my_tuple[0]`, second item with `my_tuple[1]` and so on.

It is important to note that the index of tuple items starts from 0, not 1 (similar to lists).

```
my_tuple = ('p','e','r','m','i','t')

print(my_tuple[0])   # Output: 'p'
print(my_tuple[5])   # Output: 't'
```

The index must be an integer, so we cannot use float or other types. This will result in `TypeError`.

Here, `my_tuple` has 6 items. If you try to access the 7th item using `my_tuple[6]`, you will get `IndexError`.

Negative Indexing

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

```
my_tuple = ('p','e','r','m','i','t')
print(my_tuple[-1])  # Output: 't'
print(my_tuple[-6])  # Output: 'p'
```

Here, `my_tuple` contains 6 items. In this case,

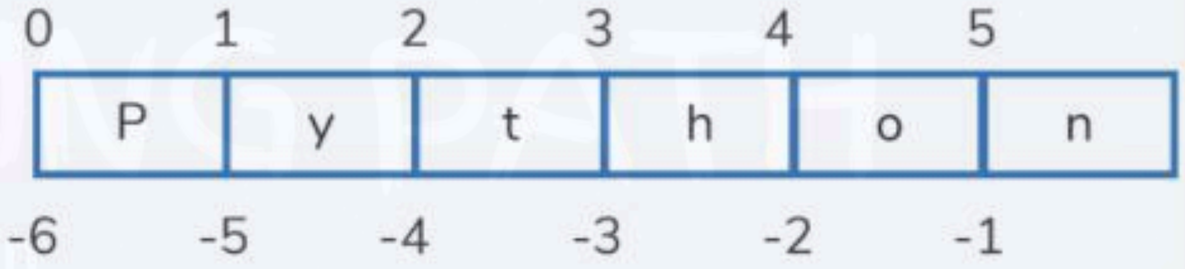
- Both `my_tuple[0]` and `my_tuple[-6]` gives us the first element `'p'`
- Both `my_tuple[1]` and `my_tuple[-5]` gives us the second element `'e'`
- Both `my_tuple[5]` and `my_tuple[-1]` gives us the last element `'t'`

Slicing

In the previous few examples, we learned to access an item from a tuple. Now, we will learn to access a range of items. This is done by using the slicing operator :

```
my_tuple = ('P','y','t','h','o','n')

# elements from 2nd to 4th
print(my_tuple[1:4])  # ('y', 't', 'h')

# elements from beginning to 2nd
print(my_tuple[:-4])  # ('P', 'y')

# elements 6th to end
print(my_tuple[5:])   # ('n',)

# elements from the beginning to the end
print(my_tuple[:])
# Output: ('P', 'y', 't', 'h', 'o', 'n')
```

Slicing can be best visualized by considering the index to be between the elements as shown below.



So, if we want to access a range, we need two indexes that will slice that portion from the tuple.

Changing Tuple Elements

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once it has been assigned.

```
my_tuple = (99, 2, 3)
my_tuple[0] = 1

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

However, if a tuple contains elements that are mutable (like lists), it can be changed.

```
my_tuple = (1, 2, ['z', 'b'])
my_tuple[2][0] = 'a'

print(my_tuple)  # Output: (1, 2, ['a',
'b'])
```

Here, the third item of `my_tuple` is a list. We can change this list's items.

Using + and * Operators

We can use the `+` operator to combine two tuples. This is also called concatenation.

If you need to repeat items of a tuple, you can use `*` operator.

```
odd = (1, 3, 5)

print(odd + (9, 7, 5))
# Output: (1, 3, 5, 9, 7, 5)

letters = ('a', 'b')

print(letters * 3)
# Output: ('a', 'b', 'a', 'b', 'a', 'b')
```

These operations are possible because we are not changing the original tuples. It's not possible. Instead, we are creating new tuples by doing these operations.

Deleting a Tuple

You cannot delete elements of a tuple.

Let's try to delete elements of a tuple using the `del` statement.

```
my_tuple = (1, 3, 4)

# trying to delete the first element
del my_tuple[0]

# TypeError: 'tuple' object doesn't support
item deletion
```

However, you can delete the tuple itself.

```
my_tuple = (1, 3, 4)
del my_tuple
```

Python Tuple Methods

Methods that add or remove items from tuples are not available. Following two methods are the only methods available for tuples.

Method	Description
`count(x)`	Returns the number of items that is equal to `x`
`index(x)`	Returns the index of the first item that is equal to `x`

```
my_tuple = ('a','p','p','l','e')

result = my_tuple.count('p')
print(result)      # Output: 2

result = my_tuple.index('p')
print(result)      # Output: 1
```

Other Tuple Operations

You can check whether an item exists in a tuple or not using the `in` keyword.

```
my_tuple = ('a','p','p','l','e',)

print('a' in my_tuple) # Output: True
print('b' in my_tuple) # Output: False
```

Iterating through a Tuple

Using a `for` loop we can iterate through each item in a tuple.

```
for name in ('John','Kate'):
    print("Hello", name)
```

Output

```
Hello John
Hello Kate
```