Python Programming Language

Prepared by: Mohamed Ayman

Algorithm Engineer at Valeo

Deep Learning Researcher and Teaching Assistant

at The American University in Cairo (AUC)

spring 2020

sw.eng.MohamedAyman@gmail.com

facebook.com/cs.MohamedAyman

linkedin.com/in/cs-MohamedAyman

github.com/cs-MohamedAyman

codeforces.com/profile/Mohamed_Ayman

Lecture 9

Tuples

Course Roadmap

Part 2: Python Collections and Strings

Lecture 7: Strings

Lecture 8: Lists

Lecture 9: Tuples

Lecture 10: Dictionaries

Lecture 11: Sets

Lecture 12: Numbers

Lecture Agenda

We will discuss in this lecture the following topics

1- Introduction to Tuple

2- Basic Tuple Operations

3- Tuple Comprehension

4- Multi-dimensional Tuple

5- Built-in Tuple Functions


Let's
START UP

Lecture Agenda

Section 1: Introduction to Tuple

Section 2: Basic Tuple Operations

Section 3: Tuple Comprehension

Section 4: Multi-dimensional Tuple

Section 5: Built-in Tuple Functions

Introduction to Tuple

- A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The main difference between the tuples and the lists is that the tuples cannot be changed unlike lists. Creating a tuple is as simple as putting different comma-separated values. Optionally, you can put these comma-separated values between parentheses also.
- The empty tuple is written as two parentheses containing nothing, to write a tuple containing a single value you have to include a comma, even though there is only one value. To access values in tuple, use the square brackets for slicing along with the index or indices to obtain the value available at that index.

```
x = ()
x = (3, )
x = (4, 6, 1, 8, 2, 5, 3)
x = ('b', 'r', 'w', 'n', 'a')
x = ('python', 'php', 'java', 'c++')
```

Python Tuples

- Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified. Tuples are used to write-protect data and are usually faster than list as it cannot change dynamically.
- It is defined within parentheses ( ) where items are separated by commas. The main difference between lists and tuples is Lists are enclosed in brackets [ ] and their elements and size can be changed, while tuples are enclosed in parentheses ( ) and cannot be updated. Tuples can be thought of as read-only lists.
- Tuples are immutable, and usually, they contain a sequence of heterogeneous elements that are accessed via unpacking or indexing (or even by attribute in the case of named tuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list. Tuples are immutable and hence they do not allow deletion of a part of it.

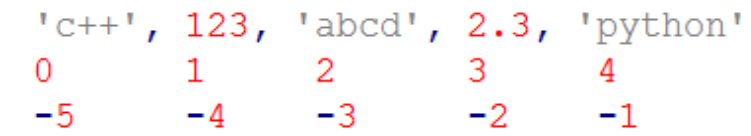
Python Tuples

Example:

```
x = ('c++', 123, 'abcd', 2.3, 'python')
print(x)
print(len(x))
print(x[3])
print(x[2:4])
print(x[:4])
print(x[2:])
print(x[-1])
print(x[-4:])
print(x[:-3])
```

Output:

```
('c++', 123, 'abcd', 2.3, 'python')
5
2.3
('abcd', 2.3)
('c++', 123, 'abcd', 2.3)
('abcd', 2.3, 'python')
'python'
(123, 'abcd', 2.3, 'python')
('c++', 123)
```



Lecture Agenda

- ✔ Section 1: Introduction to Tuple
- **Section 2: Basic Tuple Operations**
- Section 3: Tuple Comprehension
- Section 4: Multi-dimensional Tuple
- Section 5: Built-in Tuple Functions

Python Tuples

- Unlike lists, tuples are immutable. This means that elements of a tuple cannot be changed once it has been assigned. But, if the element is itself a mutable datatype like list, its nested items can be changed. We can also assign a tuple to different values (reassignment).
- Tuples are immutable, which means you cannot update or change the values of tuple elements. You are able to take portions of the existing tuples to create new tuples as the following example demonstrates. Tuples are immutable and hence they do not allow deletion of a part of it. Entire tuple gets deleted by the use of del() method. Note- Printing of Tuple after deletion results to an Error.
- Removing individual tuple elements is not possible. No thing wrong with putting together another tuple with the undesired elements discarded. As discussed above, we cannot change the elements in a tuple. That also means we cannot delete or remove items from a tuple. But deleting a tuple entirely is possible using the keyword del.

Python Tuples

Example:

```
x = ('python', 2000, 'c++', 3.2, 'java')
print(x)
print(len(x))
print(x[1] + x[3])
print(x[0] + x[-1])
print(x[:2] + x[3:])
print(x[2] * 3)
print(x[1] * 3)
print(x[2:4] * 2)
print((x[1:3] + x[4:5]) * 2)
```

Output:

```
('python', 2000, 'c++', 3.2, 'java')
5
2003.2
pythonjava
('python', 2000, 3.2, 'java')
c++c++c++
6000
('c++', 3.2, 'c++', 3.2)
(2000, 'c++', 'java', 2000, 'c++', 'java')
```

'python'	2000	'c++'	3.2	'java'
0	1	2	3	4
-5	-4	-3	-2	-1

Updating Tuples

Example:

```
x = (3, 'abc',5.2, 6, 1.7, 'ijk')
print(x)
x = x[:5] + ('def', )
print(x)
x = (8.2, 7) + x[2:]
print(x)
x = x[:4] + ('were', 9.4, 'the') + x[5:]
print(x)
x = x[:3] + ('you', ) + x[7:]
print(x)
x = x[:2] + x[-2:]
print(x)
```

Output:

```
(3, 'abc', 5.2, 6, 1.7, 'ijk')

(3, 'abc', 5.2, 6, 1.7, 'def')

(8.2, 7, 5.2, 6, 1.7, 'def')

(8.2, 7, 5.2, 6, 'were', 9.4, 'the', 'def')

(8.2, 7, 5.2, 'you', 'def')

(8.2, 7, 'you', 'def')
```

Updating Tuples

Example:

```
x = (3, 'abc', 5.2, 6, 1.7, 'ijk')

print(x)

del x[2]
```

Output:

```
(3, 'abc', 5.2, 6, 1.7, 'ijk')

Traceback (most recent call last):
  File "main.py", line 3, in <module>
    del x[2]
TypeError: 'tuple' object doesn't support item deletion
```

```
x = (3, 'abc', 5.2, 6, 1.7, 'ijk')

print(x)
del x
print(x)
```

```
(3, 'abc', 5.2, 6, 1.7, 'ijk')

Traceback (most recent call last):
  File "main.py", line 4, in <module>
    print(x)
NameError: name 'x' is not defined
```

Why Use Tuples?

- Tuples typically store heterogeneous data, similar to how lists typically hold homogeneous data. It's not a hard-coded rule but simply a convention that some Python programmers follow. Because tuples are immutable, they can be used to store different data about a certain thing.
- Tuples are processed faster than lists. If you are creating a constant set of values that won't change, and you need to simply iterate through them, use a tuple.
- The sequences within a tuple are essentially protected from modification. This way, you won't accidentally change the values, nor can someone misuse an API to modify the data. (An API is an application programming interface. It allows programmers to use a program without having to know the details of the whole program). Tuples are used in string formatting, by holding multiple values to be inserted into a string.
- For example, a contact list could conceivably stored within a tuple; you could have a name and address (both strings) plus a phone number (integer) within on data object. The biggest thing to remember is that standard operations like slicing and iteration return new tuple objects. In my programming, I like use lists for everything except when I don't want a collection to change. It cuts down on the number of collections to think about, plus tuples don't let you add new items to them or delete data. You have to make a new tuple in those cases.

Advantages of Tuple over List

- Since tuples are quite similar to lists, both of them are used in similar situations as well. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:
- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) data types. Since tuples are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible. If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected. Tuples can be used as keys for dictionaries. Honestly, I don't think I've ever used this, nor can I think of a time when you would need to. But it's there if you ever need to use it.
- Tuples are used in string formatting, by holding multiple values to be inserted into a string.

Lecture Agenda

- ✔ Section 1: Introduction to Tuple
- ✔ Section 2: Basic Tuple Operations
- **Section 3: Tuple Comprehension**
- Section 4: Multi-dimensional Tuple
- Section 5: Built-in Tuple Functions

Tuple Comprehension

- Using generator comprehensions to initialize tuples is so useful that Python actually reserves a specialized syntax for it, known as the tuple comprehension. A tuple comprehension is a syntax for constructing a tuple, which exactly mirrors the generator comprehension syntax:

```
(<expression> for <var> in <iterable> {if <condition>})
```

- For example, if we want to create a tuple of square-numbers, we can simply write:

```
x = (i**2 for i in range(7))
print(x)
```

```
(0, 1, 4, 9, 16, 25, 36)
```

- This produces the exact same result as feeding the tuple function a generator comprehension. However, using a tuple comprehension is slightly more efficient than is feeding the tuple function a generator comprehension.

Tuple Comprehension

- Generate a tuple with zeros

```
z = (0 for i in range(6))
print(z)                                    (0, 0, 0, 0, 0, 0)
z = (0) * 6
print(z)                                    (0, 0, 0, 0, 0, 0)
```

- Let's appreciate how economical tuple comprehensions are.
- The following code stores words that contain the letter 'o', in a tuple:

```
words = ('python', 'like', 'you', 'mean', 'it')
res = ()
for i in words:
    if 'o' in i:
        res.append(i)
```

- This can be written in a single line, using a tuple comprehension:

```
words = ('python', 'like', 'you', 'mean', 'it')
res = (i for i in words if 'o' in i)
print(res)                                  ('python', 'you')
```

Tuple Comprehension

Example:

```
# Input tuple
x = (int(i) for i in input().split())
print(x)
x = tuple(map(int, input().split()))
print(x)

# Copy the values of x in y
y = (i for i in x)
y = x[:]

# Updating tuple x and y
x[0] = 2.7
y[1] = 3.2
print(x)
print(y)

# Reverse x, y
print(x[::-1])
print(y[::-1])
```

Output:

```
(7, 8, 9, 6, 5, 4)
(7, 8, 9, 6, 5, 4)

(2.7, 8, 9, 6, 5, 4)
(7, 3.2, 9, 6, 5, 4)

(4, 5, 6, 9, 8, 2.7)
(4, 5, 6, 9, 3.2, 7)
```

Lecture Agenda

- ✔ Section 1: Introduction to Tuple
- ✔ Section 2: Basic Tuple Operations
- ✔ Section 3: Tuple Comprehension
- **Section 4: Multi-dimensional Tuple**
- Section 5: Built-in Tuple Functions

Multi-dimensional Tuples

- A matrix is a two-dimensional data structure. In real-world tasks you often have to store rectangular data table. The table below shows the marks of three students in different subjects.
- Such tables are called matrices or two-dimensional arrays. In python any table can be represented as a tuple of tuples (a tuple, where each element is in turn a tuple).

S.No	Student Name	Science	English	History	Arts	Maths
1	Roy	80	75	85	90	95
2	John	75	80	75	85	100
3	Dave	80	80	80	90	95

- In the above example A represents a 3*6 matrix where 3 is number of rows and 6 is number of columns.

```
A = (('Roy',80,75,85,90,95),('John',75,80,75,85,100),('Dave',80,80,80,90,95))
```

Multi-dimensional Tuples

- In python, matrix is a nested tuple. A tuple is created by placing all the items (elements) inside a circular bracket ( ), separated by commas.
- Here's a program that creates a numerical table with 3 rows and 5 columns.
- In the second examples a is a matrix as well as nested tuple where as b is a nested tuple but not a matrix.

```
# X is 2-D matrix
x = (('Roy', 80, 75, 85, 90),
     ('John', 70, 80, 75, 85),
     ('Dave', 85, 70, 80, 90))

print(x)

# Y is nested tuple but not 2-D matrix
y = (('Roy', 80, 75, 85, 90),
     ('John', 70, 80, 75),
     ('Dave', 85, 70, 80, 90))

print(y)
```

Multi-dimensional Tuples

- Create a dynamic matrix using for loop. A possible way: you can create a matrix of n*m elements by first creating a tuple of n elements (say, of n zeros) and then make each of the elements a link to another one-dimensional tuple of m elements

Example

```
n, m = 3, 4
x = ((0)*m for i in range(n))
print(x)
```

Output

```
((0, 0, 0, 0),
 (0, 0, 0, 0),
 (0, 0, 0, 0))
```

```
n, m = 3, 4
x = ()
for i in range(n):
    x += (tuple(map(int, input().split())))
print(x)
```

```
((1, 2, 3, 4),
 (5, 6, 7, 8),
 (9, 10, 11, 12))
```

Quiz

- Which of the following expressions evaluates to the tuple (1,2,3,4,5)?

- [ ] `tuple(range(1,6,1))`
- [ ] `tuple(range(1,6))`
- [ ] `range(1,6)`
- [ ] `tuple(range(5))`

- If we want to split a tuple my_tuple into two halves, which of the following uses slices to do so correctly?

  More precisely, if the length of my_tuple is 2n, i.e., even, then the two parts should each have length n. If its length is 2n+1, i.e., odd, then the two parts should have lengths n and n+1.

- [ ] `x[:len(x)//2] and x[len(x)//2:]`
- [ ] `x[0:len(x)//2-1] and x[len(x)//2:len(x)]`
- [ ] `x[0:len(x)//2] and x[len(x)//2+1:len(x)]`
- [ ] `x[0:len(x)//2] and x[len(x)//2:len(x)]`

Quiz Solution

- Which of the following expressions evaluates to the tuple (1,2,3,4,5)?

- [x] `list(range(1,6,1))`
- [x] `list(range(1,6))`
- [ ] `range(1,6)`
- [ ] `list(range(5))`

- If we want to split a tuple my_tuple into two halves, which of the following uses slices to do so correctly?

More precisely, if the length of my_tuple is 2n, i.e., even, then the two parts should each have length n. If its length is 2n+1, i.e., odd, then the two parts should have lengths n and n+1.

- [x] `x[:len(x)//2] and x[len(x)//2:]`
- [ ] `x[0:len(x)//2-1] and x[len(x)//2:len(x)]`
- [ ] `x[0:len(x)//2] and x[len(x)//2+1:len(x)]`
- [x] `x[0:len(x)//2] and x[len(x)//2:len(x)]`

Quiz

- If n and m are non-negative integers, consider the tuple final tuple computed by the code snippet below.

```
init_tuple = tuple(range(1, n))
final_tuple = init_tuple * m
```

The length of this tuple depends on the particular values of n and m used in computation. Which option below correctly expresses the length of final_tuple in terms of n and m?

- ☐ `n × m`
- ☐ `n × (m - 1)`
- ☐ `n + m`
- ☐ `(n - 1) × m`

- If n is a non-negative integer, consider the tuple split_tuple computed by the code snippet below.

```
test_string = 'xxx' + ' ' * n + 'xxx'
split_tuple = tuple(test_string.split(' '))
```

The length of this tuple depends on the particular values of n used in computation. Which option below correctly expresses the length of split_tuple in terms of n?

- ☐ `3`
- ☐ `n + 1`
- ☐ `2`
- ☐ `n`

Quiz Solution

- If n and m are non-negative integers, consider the tuple final_tuple computed by the code snippet below.

```
init_tuple = tuple(range(1, n))
final_tuple = init_tuple * m
```

The length of this tuple depends on the particular values of n and m used in computation. Which option below correctly expresses the length of final_tuple in terms of n and m?

- ☐ `n × m`
- ☐ `n × (m - 1)`
- ☐ `n + m`
- ☑ `(n - 1) × m`

- If n is a non-negative integer, consider the tuple split_tuple computed by the code snippet below.

```
test_string = 'xxx' + ' ' * n + 'xxx'
split_tuple = tuple(test_string.split(' '))
```

The length of this tuple depends on the particular values of n used in computation. Which option below correctly expresses the length of split_tuple in terms of n?

- ☐ `3`
- ☑ `n + 1`
- ☐ `2`
- ☐ `n`

Lecture Agenda

- ✔ Section 1: Introduction to Tuple
- ✔ Section 2: Basic Tuple Operations
- ✔ Section 3: Tuple Comprehension
- ✔ Section 4: Multi-dimensional Tuple
- Section 5: Built-in Tuple Functions

Built-in Tuple Functions

1- len() Method

2- tuple() Method

3- max(), min() Methods

4- count() Method

5- index() Method

6- sum() Method

7- all(), any() Methods

8- enumerate() Method

len() Method

Example:

```
x = (5, 8, 2, 9, 10, 4)
print(x)
print(len(x))

x = ('python', 'c++', 'java')
print(x)
print(len(x))
```

Output:

```
(5, 8, 2, 9, 10, 4)
6

('python', 'c++', 'java')
3
```

tuple() Method

Example:

```
x = 'python'
print(x)
y = tuple(x)
print(y)

x = [5, 16, 21, 4]
print(x)
y = tuple(x)
print(y)

x = {4, 8, 9, 3, 1}
print(x)
y = tuple(x)
print(y)

x = {'b':4, 'c':8, 'a':9}
print(x)
y = tuple(x)
print(y)
```

Output:

```
python
('p', 'y', 't', 'h', 'o', 'n')

[5, 16, 21, 4]
(5, 16, 21, 4)

{4, 8, 9, 3, 1}
(4, 8, 9, 3, 1)

{'b':4, 'c':8, 'a':9}
('b', 'c', 'a')
```

max(), min() Methods

Example:

```
x = (1, 2, 6)
print(x)
print(max(x))
print(min(x))
y = (4, 2)
print(y)
print(max(y))
print(min(y))
print(max(x, y))
print(min(x, y))
```

Output:

```
(1, 2, 6)
6
1
(4, 2)
4
2
(4, 2)
(1, 2, 6)
```

max(), min() Methods

Example:

```
x = ('php', 'c++', 'java')
print(x)
print(max(x))
print(min(x))
y = ('python', 'c#')
print(y)
print(max(y))
print(min(y))
print(max(x, y))
print(min(x, y))
```

Output:

```
('php', 'c++', 'java')
php
c++

('python', 'c#')
python
c#
('python', 'c#')
('php', 'c++', 'java')
```

count() Method

Example:

```
x = ('c#', 'R', 'c#', 'R', 'c++')
print(x)
print(x.count('R'))
print(x.count('c#'))
print(x.count('c++'))
print(x.count('python'))
```

Output:

```
('c#', 'R', 'c#', 'R', 'c++')
2
2
1
0
```

index() Method

Example:

```
x = ('c#', 'c++', 'c#', 'python')
print(x)
print(x.index('c#'))
print(x.index('c++'))
print(x.index('java'))
```

Output:

```
('c#', 'c++', 'c#', 'python')
0
1
Traceback (most recent call last):
  File "main.py", line 5 in <module>
    print(x.index('java'))
ValueError: 'java' is not in tuple
```

sum() Method

Example:

```
x = (4, 6, 1, 9, 7, 3)
print(x)
print(sum(x))

x = ('python', 'java', 'c#', 'c++')
print(x)
print(sum(x))
```

Output:

```
(4, 6, 1, 9, 7, 3)
30

('python', 'java', 'c#', 'c++')
Traceback (most recent call last):
  File "main.py", line 7, in <module>
    print(sum(x))
TypeError: unsupported operand type(s)
for +: 'int' and 'str'
```

all(), any() Methods

Example:

```
x = (True, False, False)
print(all(x))
print(any(x))

x = (True, True, False)
print(all(x))
print(any(x))

x = (False, False, False)
print(all(x))
print(any(x))

x = (True, True, True)
print(all(x))
print(any(x))
```

Output:

```
False
True

False
True

False
False

True
True
```

enumerate() Method

Example:

```
x = (25, 45, 35, 15)
print(x)

e = enumerate(x)
print(type(e))
print(tuple(e))

e = enumerate(x, 3)
print(type(e))
print(tuple(e))
```

Output:

```
(25, 45, 35, 15)

<class 'enumerate'>
((0, 25), (1, 45), (2, 35), (3, 15))

<class 'enumerate'>
((3, 25), (4, 45), (5, 35), (6, 15))
```

enumerate() Method

Example:

```
x = (25, 45, 35, 15)
print(x)

for i in enumerate(x):
    print(i)


for i, j in enumerate(x):
    print(i,j)
```

Output:

```
(25, 45, 35, 15)

(0, 25)
(1, 45)
(2, 35)
(3, 15)
0 25
1 45
2 35
3 15
```

Practice

Problems

1- Implement a function which calculates the length of a tuple
2- Implement a function which converts any collections to tuple
3- Implement a function which finds the maximum value in a tuple
4- Implement a function which finds the minimum value in a tuple
5- Implement a function which counts the number of occurrences of a given item in a tuple
6- Implement a function which finds the index of a given item in a tuple
7- Implement a function which enumerates a given tuple

Built-in Tuple Functions

1- len() Method

2- tuple() Method

3- max(), min() Methods

4- count() Method

5- index() Method

6- sum() Method

7- all(), any() Methods

8- enumerate() Method

Lecture Agenda

- ✔ Section 1: Introduction to Tuple
- ✔ Section 2: Basic Tuple Operations
- ✔ Section 3: Tuple Comprehension
- ✔ Section 4: Multi-dimensional Tuple
- ✔ Section 5: Built-in Tuple Functions


DO
MORE.