Python Data Structures: Tuples

Programming for Data Science with Python

Overview

In Python, **tuples** are the objects of the class tuple that has the **constructor tuple()**.

Tuple is an **immutable** sequence data type/structure, i.e., its **contents cannot be changed** after being created.

Tuples work similarly to strings and lists:

- Use the len() function for the length of a tuple
- Use square brackets [] to access data, with the first element at index 0
- The range of indices: 0 .. len(a tuple) - 1

IMPORTANT NOTES:

What are the benefit of tuples?

- Tuples are **faster than lists**.
- If you know that some **data doesn't have to be changed**, you should **use tuples** instead of lists (because this protects your data against accidental changes.)
- Tuples can be used as **keys in dictionaries**, while lists can't.
- We generally use **tuple** for **heterogeneous (different) datatypes** and list for homogeneous (similar) datatypes.

1.1 Properties of Tuples

A tuple is an **immutable list**, i.e., a tuple cannot be changed in any way once it has been created.

A tuple is defined analogously to lists except that the set of elements is enclosed in parentheses instead of square brackets.

The rules for indices are the same as for lists. Once a tuple has been created, you can't add elements to a tuple or remove elements from a tuple.

IMPORTANT NOTES:

It is actually the comma which makes a tuple, not the parentheses:

- The parentheses are optional, except in the empty tuple case **OR** when they are needed to avoid syntactic ambiguity.

For example:

- f(a, b, c) is a function call with three arguments
- f((a, b, c)) is a function call with a 3-tuple as the sole argument.

Run the following code:

In [51]:
```
t = ("tuples", "are", "immutable")
t[0]
```

Out[51]:
```
'tuples'
```

1.3 Elements of Tuples

Index range of list elements

Forward index range of list elements: **0 .. len(list) – 1** Forward: starting from the 1st element

Backward index range of list elements: **-1 .. -len(list**) Backward: Starting from the last element

Run the following code:

In [52]:
```
t = ("tuples", "are", "immutable")
print(t[0])
print(t[-1])
print (t[-3])
print(len(t))
```

```
tuples
immutable
tuples
3
```

1.4 Constructor: tuple ([iterable])

The constructor builds a tuple whose items are the same and in the same order as iterable's items.

- Iterable may be either a sequence, a container that supports iteration, or an iterator object.
- If iterable is already a tuple, it is returned unchanged.

If no argument is given, the constructor creates a new empty tuple:().

Run the following 2 code blocks:

In [53]:
```
tuple = ("a","b","c")
print(tuple)
```

```
('a', 'b', 'c')
```

In [56]:
```
tuple_a = ([1,2,3])
print(tuple_a)
```

```
[1, 2, 3]
```

2. Create Tuples

Tuples may be constructed in a number of ways:

- Using a pair of parentheses to denote the empty tuple: ()
- Using a trailing comma for a singleton tuple: a, or (a,)
- Separating items with commas: a, b, c or (a, b, c)
- Using the tuple() built-in: tuple() or tuple(iterable)

A tuple is created by placing all the items (elements) inside a parentheses(), separated by comma. The parentheses are optional but is a good practice to write it.

Run the following 4 code blocks:

In [64]:
```
# empty tuple
# Output: ()
my_tuple = ()
print(my_tuple)
```

```
()
```

In [65]:
```
# tuple having integers
# Output: (1, 2, 3)
my_tuple = (1, 2, 3)
print(my_tuple)
```

```
(1, 2, 3)
```

In [66]:
```
# tuple with mixed datatypes
# Output: (1, "Hello", 3.4)
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
```

```
(1, 'Hello', 3.4)
```

In [85]:
```
# nested tuple
# Output: ("mouse", [B, 4, 6}, (1, 2, 3))
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

```
('mouse', [8, 4, 6], (1, 2, 3))
```

2.1 Create tuples with only ONE element

Creating a tuple with one element is a bit tricky.

Placing one element within parentheses is not enough. We must add a **trailing comma** to indicate that it is in fact a tuple.

Run the following 3 code blocks:

In [86]:
```
# only parentheses is not enough

my_tuple = ("hello")
print(type(my_tuple))
```

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

In [87]:
```
# need a comma at the end

my_tuple = ("hello",)
print(type(my_tuple))
```

```
<class 'tuple'>
```

In [93]:
```
# parentheses are optional

my_tuple = "hello",
print(type(my_tuple))
```

```
<class 'tuple'>
```

3. Access List Elements

As other sequence data types/structures, list elements can be accessed via their indices.

We can use the index operator [] to access an item in a list. **Index starts from 0**. So, a list having 5 elements will have index from 0 to 4. Trying to access an element other than this will raise an IndexError.

The index must be an integer. We can't use float or other types, this will result into TypeError.

Nested list are accessed using nested indexing [][] that is similar to index of 2-D array elements.

3.1 Access Single Elements of Tuples

Run the following 8 code blocks:

In [101]:
```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[0])
```

```
p
```

In [102]:
```
print(my_tuple[5])
```

```
t
```

In [103]:
```
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(n_tuple[0][3])
```

```
s
```

In [104]:
```
print(n_tuple[1][1])
```

```
4
```

In [105]:
```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
len(my_tuple)
```

Out[105]:
```
6
```

In [106]:
```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
print(my_tuple[-1])
```

```
t
```

In [107]:
```
print(my_tuple[-6])
```

```
p
```

In [108]:
```
my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
# Range of the indices: 0 … len(my_tuples) -1: 0 … 6
# Index must be in range
# Or you will get an ERROR: since the index is out of range
print (my_tuple[6])
```

```
---------------------------------------------------------------------------
------
IndexError                                Traceback (most recent call
last)
<ipython-input-108-a99fb7cd5e81> in <module>
      3 # Index must be in range
      4 # Or you will get an ERROR: since the index is out of range
----> 5 print (my_tuple[6])

IndexError: tuple index out of range
```

3.2 Access a slice of Tuples

Run the following 4 code blocks:

In [114]:
```
# elements 2nd to 4th
my_tuple =('p','r','o','g','r','a','m','i','z')

print(my_tuple[1:4])
```

```
('r', 'o', 'g')
```

In [115]:
```
# elements beginning to 2nd
print(my_tuple[:-7])
```

```
('p', 'r')
```

In [116]:
```
# elements 8th to end
# Output: ('i ', 'z')
print(my_tuple[7:])
```

```
('i', 'z')
```

In [117]:
```
# elements beginning to end
# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
print(my_tuple[:])
```

```
('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

4. Modify Tuples

4.1 All elements are immutable objects (integers, floats, strings, etc.)

IMPORTANT NOTES:

Tuples are immutable, i.e., they cannot be changed after being created. Any attempt to change or modify contents of tuples will lead to errors.

Run the following code block:

In [118]:
```
# Here you see that a tuple can not be modified; you get an error.
aTuple = ('Python', 'C', 'C++', 'Java', 'Scala')
aTuple[2] = 'Ruby'
```

```
---------------------------------------------------------------------
------
TypeError                                 Traceback (most recent call
last)
<ipython-input-118-66a351dac12a> in <module>
      1 # Here you see that a tuple can not be modified; you get an e
rror.
      2 aTuple = ('Python', 'C', 'C++', 'Java', 'Scala')
----> 3 aTuple[2] = 'Ruby'

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

4.2 One or more elements are mutable objects: lists, byte arrays, etc.

Tuples are **immutable**.

- This means that elements of a tuple cannot be changed once it has been assigned.
- If the element is itself a mutable datatype, like list, its nested items can be changed.

Run the following code block:

In [119]:
```
my_tuple = (4, 2, 3, [6, 5])
# An item of mutable element (list) can be changed

my_tuple[3][0] = 9
print(my_tuple)
```

```
(4, 2, 3, [9, 5])
```

4.3 Tuples can be Reassigned

Tuples are immutable.

- This means that elements of a tuple cannot be changed once it has been assigned, but an existing tuple variable can be reassigned with a brand new tuple.

Run the following 2 code blocks:

In [120]:
```
tuple_1 = (4, 2, 3, [6, 5])
print ("tuple_1: ", tuple_1)
```

```
tuple_1:  (4, 2, 3, [6, 5])
```

In [121]:
```
# Reassign a new tuple to tuple1

my_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
tuple_l = my_tuple
print("tuple_l after being reassinged: ", tuple_l)
```

```
tuple_l after being reassinged:  ('p', 'r', 'o', 'g', 'r', 'a', 'm',
'i', 'z')
```

5. Copy Tuples

5.1 Shallow Copy

Shallow copy means that only the reference to the object is copied. No new object is created.

Assignment with an = on lists does not make a copy. Instead, assignment makes the **two variables point to the same tuple** in memory.

Run the following code block:

In [122]:
```
tuple_l = "Hello"
tuple_2 = tuple_1
# Both the tuples refer to the same object, i.e., the same id value
id(tuple_1), id(tuple_2)
```

Out[122]:
```
(140329572936176, 140329572936176)
```

5.2 Deep copy

Deep copy means that a new object will be created when the copying has done.

IMPORTANT NOTES:

Tuples are immutable sequence objects. Tuples **cannot** be deep-copied.

5.3 Delete Tuples

To delete a string, using the built-in function del().

Run the following 2 code blocks:

In [123]:
```
aTuple = "Python is the best scripting language."
del (aTuple)
```

In [124]:
```
# To show that the string has been deleted, let's print it
# You see you get an ERROR
print (sample_str)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
<ipython-input-124-92951fc8d1f2> in <module>
      1 # To show that the string has been deleted, let's print it
      2 # You see you get an ERROR
----> 3 print (sample_str)

NameError: name 'sample_str' is not defined
```

6. Operations on Tuples

6.1 Concatenate Tuples

Run the following code block:

In [31]:
```
tuple1 = 'Hello', # comma to indicate this is a tuple; parentheses are
tuple2 = ' ',
tuple3 ='World!',

# using +
print('tuple1 + tuple2 + tuple3 = ', tuple1 + tuple2 + tuple3)
```

```
tuple1 + tuple2 + tuple3 =  ('Hello', ' ', 'World!')
```

6.2 Replicate tuples

Using * to replicate a tuple

Run the following code block:

In [32]:
```
Tuple1 = "Hello",
replicatedTuple = tuple1 * 3
print (replicatedTuple)
```

```
('Hello', 'Hello', 'Hello')
```

6.3 Test elements with "in" and "not in"

Run the following 3 code blocks:

In [33]:
```
aTuple = (2, 4, 6, "This", "is", "a", "tuple")
print (2 in aTuple)
```

```
True
```

In [34]:
```
print ('a' in aTuple)
```

```
True
```

In [35]:
```
print ("This is" in aTuple)
```

```
False
```

6.4 Compare Tuples: <, >, <=, >=, ==, !=

Run the following code block:

In [36]:
```
Tuple1 = "Hello World!"
tuple2 = "hello world!"
print (tuple1 == tuple2)
```

```
False
```

6.5 Iterate a tuple using for loop

Run the following 3 code blocks:

In [37]:
```
tuple1 = ("This", "is", 1, "book")
for i in tuple1:
   print (i)
```

```
This
is
1
book
```

In [38]:
```
Tuple1 = ("This", "is", 1, "book")
for i in tuple1:
   print (i, end="")
```

```
Thisis1book
```

In [39]:
```
Tuple1 = ("This", "is", 1, "book")
for i in tuple1:
    print(i, end="\n")
```

```
This
is
1
book
```

7. Class Tuple

7.1. count()

count(x) returns the number of elements of the tuple that are equal to (x)

Run the following code block:

In [125]:
```
my_tuple = ('a', 'p', 'p', 'l', 'e',)
# Count
print(my_tuple.count('p'))
```

```
2
```

7.2 index (x)

index(x) returns the index of the first element that is equal to (x)

Run the following code block:

In [126]:
```
# Index
my_tuple=('a','p','p','l','e',)
print(my_tuple.index('l'))
```

```
3
```