Department of Computer Science and Engineering
PES University, Bangalore, India

Lecture Notes
Python for Computational Problem Solving
UE23CS151A

Lecture #29
Tuple and it's Operations

By,
Prof. Sindhu R Pai,
Anchor, PCPS - 2023
Assistant Professor
Dept. of CSE, PESU

Verified by,
PCPS Team - 2023

Many Thanks to
Dr. Shylaja S S (Director, CCBD and CDSAML Research Centers, Former Chairperson, CSE, PES University)
Prof. Chitra G M(Asst. Prof, Dept. of CSE, PCPS Anchor – 2022)

Tuple in Python

Introduction

Tuple is a **Non-primitive Linear Data Structure**. It is a **collection of different types of values which are unmodified**. Let us try to understand why list is required through an example.

Consider the scenario where every user wants to store the city name and the pin code of the area he/she belongs to, together in one variable. If the user is able to modify the pin code of area in a city according to their requirement, we may end up with many areas with same pin-code or there will be uniformity in the pin code representation. If we do want this to happen then we may have to keep the **values unmodified.** This is where the concept of tuple comes in to picture.

Characteristics of Tuples:

- It has **0 or more elements.**
- It allows duplicate elements within it.
- There is no **name for each element** in a tuple separately.
- Elements are accessed using **indexing operation or by subscripting**.
- Tuple is **indexable** - **Index always starts with 0**. This is known as **zero based indexing. Negative indices are also supported.**
- Tuple is **Immutable** - Once created, we cannot change the number of elements. It cannot grow or shrink – no append, no insert, no remove, no clear, no pop. Using these results in AttributeError. Direct Assignment using the index results in TypeError.

```
>>> numbers
(12, 78, 33, 32.7, 11.9, 83, 78)
>>> numbers.append(104)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> numbers[2] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

- Tuple is **iterable –** Can get one element at a time.
- **Elements** in the tuple can be **heterogeneous** in nature.

Syntactically, elements of the tuple must be inside **parentheses – (** and **)** and **all the elements must be separated by a comma** between each of them.

A. Create a tuple of numbers

```
numbers = (12, 78, 33, 32.7, 11.9, 83, 78)      #Duplicates allowed
        #tuple has heterogeneous type of elements in it. As of now int and float
print(type(numbers))   #  <class 'tuple'>
```

Note: Empty tuple can be created using () or tuple()

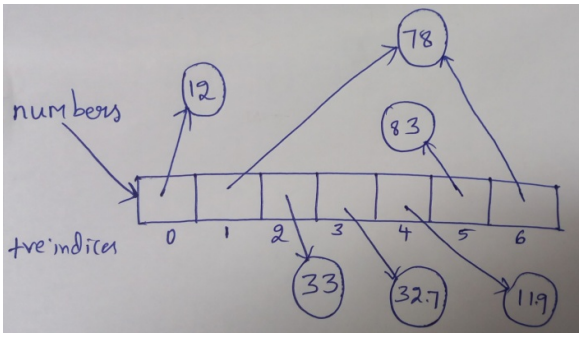
Think about creation of a tuple with a single element!!

```
>>> numbers = ()
>>> type(numbers)
<class 'tuple'>
>>> numbers = tuple()
>>> type(numbers)
<class 'tuple'>
>>> numbers = (10)  #is it a tuple with one element?
>>> type(numbers)
<class 'int'>            #No
>>> numbers = (10,)  #is it a tuple with one element? Use a trailing comma after the element to represent it as a tuple with a single element.
>>> type(numbers)
<class 'tuple'>   #yes
>>> numbers = 10,    #is it a tuple with one element?
>>> type(numbers)
<class 'tuple'>   #yes
>>> t = tuple(23)    #usage of tuple() to create a tuple with single element
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> t = tuple(23,)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> t = tuple((23,))     # value must be inside ( and ) separated by comma(s)
>>> t
```

```
(23,)
>>>
```

B. Diagrammatic representation of a tuple

numbers = (12, 78, 33, 32.7, 11.9, 83, 78)



C. Accessing the elements of the tuple

The index begins with 0. To access 33, we can use **numbers[2].** If we want to use negative index, -1 is the index for the last element of the list. To access 33, we can say, **numbers[-5]. Max value of index is length of the tuple – 1. Accessing outside this index results in Index Error. The last element of the list** can be accessed **using -1 as the index or len(numbers) – 1.**

```
>>> numbers = (12, 78, 33, 32.7, 11.9, 83, 78)
>>> numbers[2]
33
>>> numbers[-2]
83
>>> numbers[len(numbers)]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> numbers[len(numbers)-1]
78
>>> numbers[-1]
78
>>>
```

D. **Common functions that can be applied are len(), max(), min(), sum().** These functions are self explanatory

```
>>> numbers = [12, 78, 33, 32.7, 11.9, 83, 78]
>>> len(numbers)
7
>>> max(numbers)
83
>>> min(numbers)
11.9
>>> sum(numbers)
328.6
```

E. Common operators that can be applied are +, *, in, not in, slicing operator(:) within index operator([]) and relational operators

- **+ ->** The **Concatenate operation** is used to merge two tuples and creates new tuple.
- ***->** The **Repetition operation** allows multiplying the tuple n times and create a new tuple.
- **in and not in ->** Used to **find whether the particular element exists in the tuple or not**.

```
>>> tpl1 = (13,44,55,12,44,"pes", "university")
>>> tpl2 = ("chocolate", 12,55,88)
>>> tpl1 + tpl2
(13, 44, 55, 12, 44, 'pes', 'university', 'chocolate', 12, 55, 88)
>>> tpl1 * 2
(13, 44, 55, 12, 44, 'pes', 'university', 13, 44, 55, 12, 44, 'pes', 'university')
>>> tpl1 * 0
()
>>> 2 * tpl2
('chocolate', 12, 55, 88, 'chocolate', 12, 55, 88)
>>> "pes" in tpl1
True
>>> "pes" not in tpl1
False
>>>
```

Try it ! ->

tpl1 + 4
tpl1+(4,)

- **Relational operators ->** Compares the corresponding elements until a mismatch or one or both ends. Works on tuple same as lists in python.

```
>>> tpl1 = (23,44,11,77)
>>> tpl2 = (23,44,11,77)
>>> tpl1 == tpl2
True
>>> tpl3 = (23,44,12)
>>> tpl1 < tpl3
True
>>> tpl4 = (23,44,12,54)
>>> tpl4 < tpl1
False
>>> tpl4 < tpl3
False
>>>
```

- **Slicing operator [:] ->** Used to **create a new tuple based on the indices of the existing** tuple. But if you create a new tuple having all the elements in the existing tuple, using the slice operator, doesn't create a copy of the original tuple.

  Given **tpl1 = (23,12,99,67,45,23)**

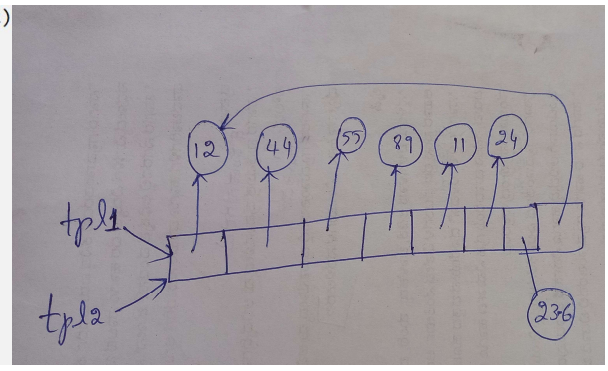
tpl2 = tpl1[::] # doesn't create a copy of tpl1. Works the same way as tpl2 = tpl1 as it is immutable

print(id(tpl1))

print(id(tpl2)) # These two id values will be same.

This is due to the immutable property of tuple

```
>>> tpl1 = (12,44,55,89,11,24, 23.6, 12)
>>> tpl2 = tpl1[::]
>>> tpl1
(12, 44, 55, 89, 11, 24, 23.6, 12)
>>> tpl2
(12, 44, 55, 89, 11, 24, 23.6, 12)
>>> id(tpl1)
3104982947696
>>> id(tpl2)
3104982947696
>>> tpl3 = tpl1[0:5]
>>> tpl3
(12, 44, 55, 89, 11)
>>> id(tpl3)
3104983233328
```



But if you try to create a copy of few elements only using slicing operator, then it creates a new copy. Check the id of tpl1 and tpl3 in the above code. It is different.

Specific Functions on Tuples: The number of functions on tuple is much lesser than the number of functions on lists. Use dir() function to **display the list of functions a type supports**. Can use help() on any of these to know its job and it' usage. Samples are shown below.

```
>>> dir(tuple)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__dir__', '__doc__', '
__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getstate__'
, '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '__setattr__'
, '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']
>>>
```

```
>>> help(tuple.count)
Help on method_descriptor:

count(self, value, /)
    Return number of occurrences of value.

>>>
```

```
>>> help(tuple.index)
Help on method_descriptor:

index(self, value, start=0, stop=9223372036854775807, /)
    Return first index of value.

    Raises ValueError if the value is not present.

>>>
```

index(): Return first index of value. Raises ValueError if the value is not present.

```
>>> tpl1 = (23, 55, 11, 88, 55)
>>> tpl1.index(55)
1
```

```
>>> tpl1.index(909)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: tuple.index(x): x not in tuple
>>>
```

count(): Returns the number of occurrences of value.

```
>>> tpl1 = (23, 55, 11, 88, 55, 123, 55, 23, 55, 555, 12)
>>> tpl1.count(55)
4
>>> tpl1.count(505)   %505 is not there
0
>>>
```

There are many functions available for list type as shown in the result of dir() function above. **If you want to find whether some function is supported by list or not, use the in operator to do that.**

Example:

```
>>> 'append' in dir(tuple)
False
>>> 'apend' in dir(tuple)
False
>>>
```

TRY IT!

- **Can we convert the list type of variable to tuple type and create a new variable? If yes, how? - Use tuple()**
- **If you want to clear the elements of the tuple, which function helps you?**
- **If you want to get the sorted list of elements from the tuple, which function helps? – sorted() from built-ins**

Few points to note:

- If the elements are separated by a comma in between, automatically the variable at the LHS of assignment operator is of **type tuple.** When you assign this to another variable, another **variable is also of type tuple**. But if you assign it to

variables using named variables at LHS, **unpacking of the iterable happens first and then assignment takes place. Hence, elements are stored in respective variables and the type is decided based on what is stored in that variable.**

```
>>> a = 1,2,3
>>> type(a)
<class 'tuple'>
>>> x,y,z = a
>>> type(x)
<class 'int'>
>>> type(y)
<class 'int'>
>>> type(z)
<class 'int'>
>>>
```

```
>>> a = 1,4,7,"sindhu", "pes"
>>> x,y,z,w,u = a
>>> type(a)
<class 'tuple'>
>>> type(y)
<class 'int'>
>>> type(w)
<class 'str'>
>>> w
'sindhu'
>>> y
4
```

```
>>> a
(1, 4, 7, 'sindhu', 'pes')
>>> w,x,y,z,u,v = a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected 6, got 5)
>>> w,x,y,z = a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 4)
>>>
```

- We may also use unnamed tuples while assigning to # of variables. The corresponding elements are assigned. The variable a becomes 11 and b becomes 22.

```
>>> a,b = 11,22
>>> a
11
>>> b
22
>>> type(a)
<class 'int'>
>>> type(b)
<class 'int'>
>>>
```

- Swapping of two variables can be done easily using the below code.

```
>>> a = 100
>>> b = 400
>>> a
100
>>> b
400
>>> a,b = b,a  #right side of assignment operator is evaluated completely first and then unpacking h
appens and a and b variables will get the new values
>>> a
400
>>> b
100
>>> ■
```

-END–