எங்கள் வாழ்வும் எங்கள் வளமும்
மங்காத தமிழ் என்று சங்கே முழங்கு ... *புரட்சிக்கவி*

NOTICE

www.DataScienceInTamil.com

Day 12 and 13 - Batch 3 - Python Language

Chapter 09 – Tuple (Collection Data Type)

Day 12- Batch 3 - Tuple- Python Collection Data Type - Part 1
https://youtu.be/hlvb4Dtv4j8

Day 13- Batch 3 - Tuple- Python Collection Data Type - Part 2
https://youtu.be/oH5-XNAxW6U

To watch the recorded Python and

Data Science videos in YouTube:

https://www.youtube.com/channel/UCTCMjShTpZg96cXloCO9q1w

Official Website:

https://DataScienceInTamil.com/

மேலும் முக்கிய கேள்விகள் பதில்களுக்கு :

https://www.DatascienceInTamil.com/#faq

To join DataScienceInTamil Telegram group:

இந்த குழுவில் உங்கள் நண்பர்களை இணைக்க விரும்பினால் அதற்கான லிங்க்

https://t.me/joinchat/lUZEsr-zidpjZjEx

To Join the class, please fill the form :

https://forms.gle/QFpLHwAoinFaX2cE6

Join Zoom Meeting (From Sep 26 2022 to Oct 26 2022)

https://us06web.zoom.us/j/88900302653?pwd=MVBFUlhqTTE1LzFFRUVpTzZ2S1Vsdz09

Meeting ID: 889 0030 2653

Passcode: 1234

Monday through Friday 8 PM to 10 PM IST (From Sep 26 2022 to Oct 26 2022)

We support open-source products to spread Technology to the mass.

- This is completely a FREE training course to provide introduction to Python language
- All materials / contents / images/ examples and logo used in this document are owned by the respective companies / websites. We use those contents for FREE teaching purposes only.

- We take utmost care to provide credits whenever we use materials from external source/s. If we missed to acknowledge any content that we had used here, please feel free to inform us at info@DataScienceInTamil.com.
- All the programming examples in this document are for FREE teaching purposes only.

Thanks to all the open-source community and to the below websites from where we take references / content /code example, definitions, etc., please use these websites for further reading:

- Book : Python Notes For Professionals
- https://www.w3schools.com
- https://www.geeksforgeeks.org
- https://docs.python.org
- https://www.askpython.com
- https://docs.python.org
- https://www.programiz.com

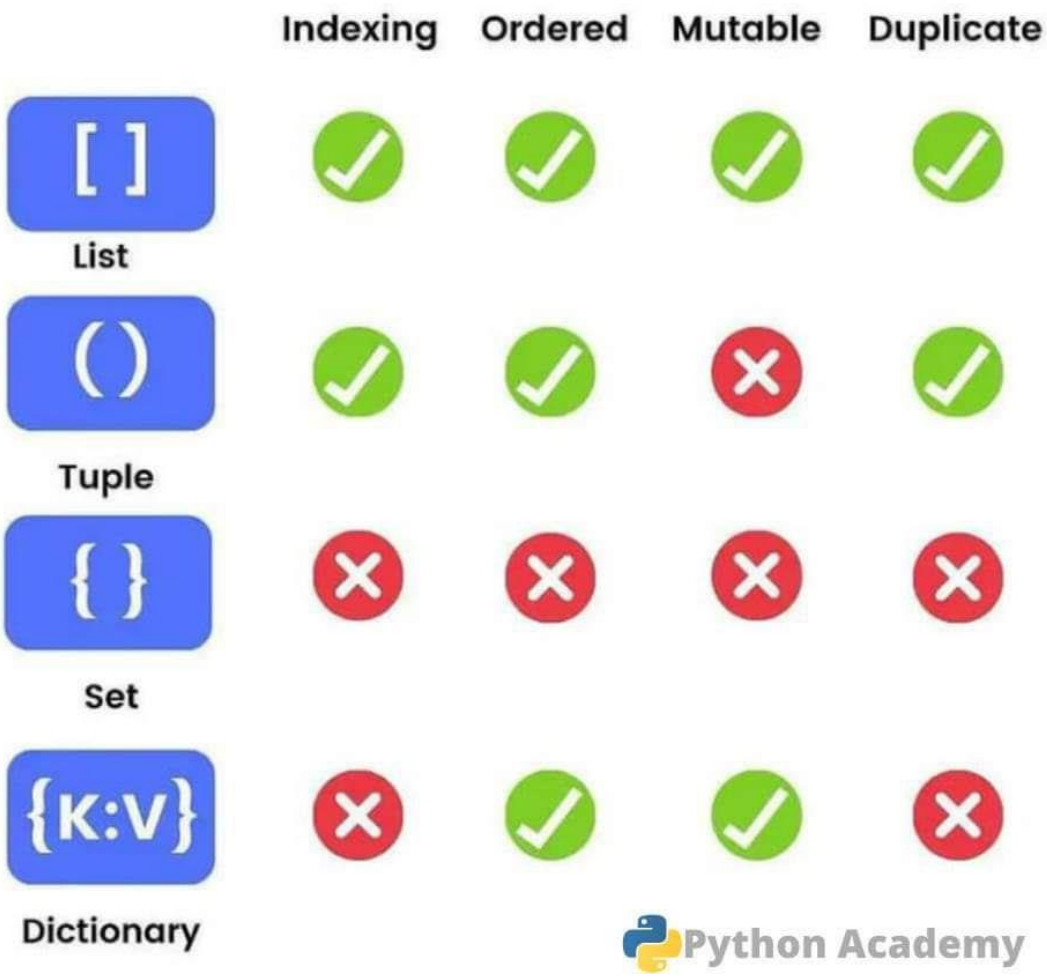
- https://www.programiz.com/
- https://www.openriskmanagement.com/
- https://pynative.com/python-sets/
- https://www.alphacodingskills.com/
- https://codedestine.com/
- https://appdividend.com/
- https://freecontent.manning.com/

Topic : Tuple

What to cover today

1. Properites of Tuple
2. Built-in Tuple methods
3. Tuples are Immutable
4. Hashable
5. Unhashable
6. Data Types and its hash ids
7. getsize()
8. Declaration of Tuple
9. Tuple Are Element-wise Hashable and Equatable
10. Packing and Unpacking Tuples
11. Catch-all variable (see Unpacking Iterables) using *
12. Convert a list into tuple
13. Tuple concatenation Use + operator to concatenate two tuples
14. __eq__() to check hash values are equivalent (hash(), __hash__()
15. Immutable date types inside tuple and how it behaves
16. Indexing Tuples
17. Indexing with negative numbers
18. Reversing Elements

Python Data Structures



There are four collection data types in the Python programming language:

Collection is heterogeneous (means it accepts any data type, where as array is homogeneous)

- **List** is a collection which is ordered and changeable. Index is possible, Allows duplicate members.not hasable ..no hash-id/ unhashable type (Mutable)
-
- **Tuple** is a collection which is ordered and unchangeable. Index is possible, . Allows duplicate members. hasable ..hash-id/ hashable type (Immutable)
-
- **Set** is a collection which is unordered and unindexed. No duplicate members, Index is NOT possible
- **Dictionary** is a collection which is ordered, changeable, and indexd(only key can be indexed – after changing the dict to list. No duplicate key members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

Tuple's methods

Tuples are enclosed in parentheses ( ) and cannot be updated. Tuples are immutable has hashid

Python has two built-in methods that you can use on tuples.

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

Python differentiates between ordered sequences (Ex List , Tuples and dict) and Unordered collections (Ex set).

Tuples are Immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be

stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

An ordered collection of n values of any type (n >= 0).

Supports indexing; immutable; **hashable if all its members are hashable/ immutable**

Most **python** objects (booleans, integers, floats, strings, bytes, complex, frozen set and tuples) are **immutable**

All immutable are hashable

List, dict, set , bytearray are mutable ie **unhashable**

Python is for Server Programming. An object is said to be **hashable (immutable)** if it has a hash value that remains the same during its lifetime. It has a __hash__() method and it can be compared to other objects

This shows that any function is hashable as it has a hash value that remains same over its lifetime.

```
a = (1, 2, 3)
# b = ('a', 1, 'python', (1,2),{'Name': "AAFI"}) #(This is unhashable since this tuple has a dictionary item that is mutable

b = ('a', 1, 'python', (1,2))# #(This is hashable since this tuple's items immutable

print (b.__hash__())
print (id(b))
print (hash(b))
```

output

```
-4015067422169779751
2110541814824
-4015067422169779751
```

Hashable / unhashable

An object is *hashable* if it has a hash value which never changes during its lifetime (it is applicable only to (basic datatypes) **booleans, integers, floats, strings, bytes, complex frozen set and tuples** (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() method). Hashable objects which compare equal must have the same hash value.

__eq__()

```
b = ('a', 1, 'python', (1,2))# #(This is hashable since this tuple's items   immutable
c = ('a', 1, 'python', (1,2))# #(This is hashable since this tuple's items   immutable
print (b.__hash__())
print (id(b))
print (hash(b))

print("===============")
print (c.__hash__())
print (id(c))
print (hash(c))

print("===============")
if b.__eq__(c):
    print("yes, both objects are equal")
```

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries or set) are not. Objects which are instances of user-defined classes are hashable by default (but the hash and id values keep changes for every execution). They all compare unequal (except with themselves), and their hash value is derived from their id().

ie, to find the hash value of the instance/object of the class use, id(instance). Or hash(instance),

```
class fruti:
    pass

c = fruti()

print(hash((c)))
print(id(c))
```

output

106164095970
1698625535520

bytes datatype has its own hash id- see below

Immutable Objects

Most python objects (booleans, integers, floats, strings, and tuples) are immutable. This means that after you create the object and assign some value to it, you can't modify that value.

Definition An immutable object is an object whose value *cannot* change.

What does this mean behind the scenes, in computer memory? An object created and given a value is assigned some space in memory. The variable name bound to the object points to that place in memory. Figure 1 shows the memory locations of objects and what happens when you bind the same variable to a new object using the expressions `a = 1` and then `a = 2`. The object with value 1 still exists in memory, but you've lost the binding to it.

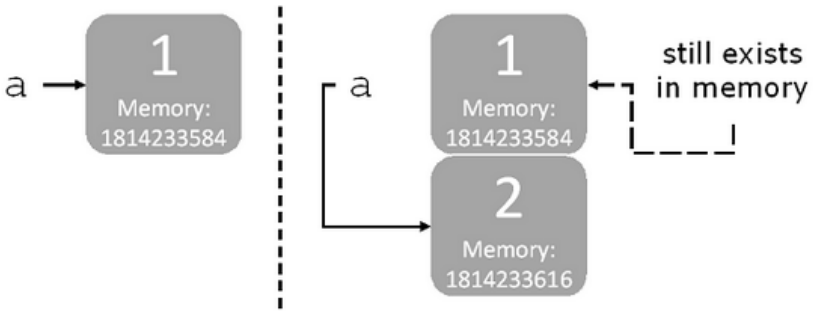
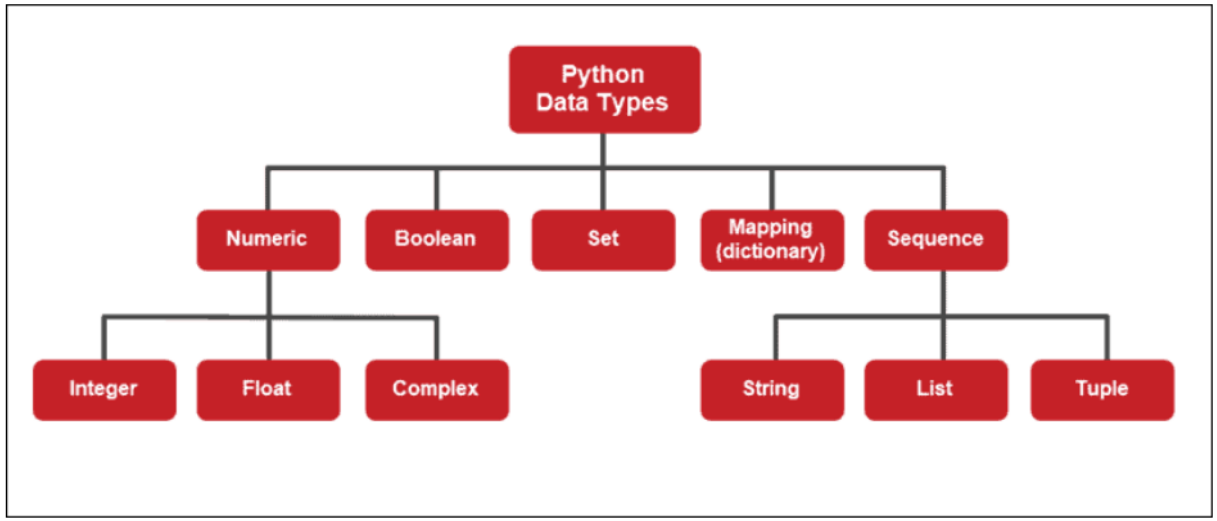


Figure 1. The variable named `a` is bound to an object with value 1 in one memory location. When the variable named `a` is bound to a different object with value 2, the original object with value 1 is still in memory, but you can't access it anymore through a variable.

Data Types and its hash ids




```
int1 = 11
print("Hash id of int : ", hash(int1))
int1 = 525625256
print("Hash id of int : ", hash(int1))
print("===============")

float1 = 44.5
print("Hash id of float  : ", hash(float1))
float1 = 5685.52
print("Hash id of float  : ", hash(float1))
print("===============")
```

```
string1 = "String"
print("Hash id of string : ", hash(string1))
print("===============")

bool1 = True
print("Hash id of bool True  : ", hash(bool1))
bool1 = False
print("Hash id of bool False  : ", hash(bool1))
print("===============")

bytes1 = b"Melcose"
print("Hash id of bytes : ", hash(bytes1))
print("===============")

complex1 = 3+5j
print("Hash id of complex : ", hash(complex1))
print("===============")

# byteArray1 = bytearray(10) #bytearray mutable
# print("Hash id of bytesArray : ", hash(byteArray1))
```

output

```
Hash id of int :  11
Hash id of int :  525625256
===============
Hash id of float  :  1073741868
Hash id of float  :  1032811102
```

==========
Hash id of string : 2138495858
==========
Hash id of bool True : 1
Hash id of bool False : 0
==========
Hash id of bytes : 351843533
==========
Hash id of complex : 5000018
==========
How to create value of byte datatype and byte datatype has hash id

```
a = bytes("Muthu",'UTF8')
print(a)
print(hash(a))

a = bytes("Muthu",'UTF16')
print(a)
print(hash(a))
```

output

```
b'Muthu'
444063461
b'\xff\xfeM\x00u\x00t\x00h\x00u\x00'
```

-1170558228

How to create value of bytearray datatype and DOESNOT hash id
Bytearray is unhashable / it does not have hash id

```
array1 = bytearray("Data Science", 'utf-8') # UTF8 means, each char represented by 8 bits / 1byte
array2 = bytearray("Numpy", 'utf-16') # UTF6 means, each char represented by 16 bits / 2 bytes
array3 = bytearray("Numpy", 'utf-32')

print(array1)
print(array2)
print(array3)
```

output

```
bytearray(b'Data Science')
bytearray(b'\xff\xfeN\x00u\x00m\x00p\x00y\x00')
bytearray(b'\xff\xfe\x00\x00N\x00\x00\x00u\x00\x00\x00m\x00\x00\x00p\x00\x00\x00y\x00\x00\x00')
```

Getsize()

```
import sys
array1 = bytearray("அ", 'utf-8') # UTF8 means, each char represented by 8 bits / 1byte
array2 = bytearray("அ", 'utf-16') # UTF6 means, each char represented by 16 bits / 2 bytes
array3 = bytearray("அ", 'utf-32')
```

```
print(array1)
print(sys.getsizeof(array1))
print(array2)
print(sys.getsizeof(array2))
print(array3)
print(sys.getsizeof(array3))

print("----------------------")

array1 = bytearray("ب", 'utf-8') # UTF8 means, each char represented by 8 bits / 1byte
array2 = bytearray("ب", 'utf-16') # UTF6 means, each char represented by 16 bits / 2 bytes
array3 = bytearray("ب", 'utf-32')

print(array1)
print(sys.getsizeof(array1))
print(array2)
print(sys.getsizeof(array2))
print(array3)
print(sys.getsizeof(array3))
```

output

```
bytearray(b'\xe0\xae\x85')
32
bytearray(b'\xff\xfe\x85\x0b')
33
bytearray(b'\xff\xfe\x00\x00\x85\x0b\x00\x00')
37
----------------------
```

bytearray(b'\xd8\xa8')
32
bytearray(b'\xff\xfe(\x06\x00')
35
bytearray(b'\xff\xfe\x00\x00(\x06\x00\x00\x00\x00\x00')
41
==========
If tuple has only int, the hash value of the each int the same value. If tuple has any other datatype, each object / element will have its own hash id and the hash id keep changes for every execution

```
name = "Nathan"
print(hash(name))
t = (10,20,30,40.5, name, True, False, bytes("Sudha", 'utf-8'))

print("----------")
name1 = "Nathan"
print(hash(name1))

print("----------")
print(t)
print(hash(t))
for item in t:
    print(item, hash(item))
```

output

```
131839970
----------
131839970
----------
```

```
(10, 20, 30, 40.5, 'Nathan', True)
-1305745827
10 10
20 20
30 30
40.5 1073741864
Nathan 131839970
True 1
--------------
```

```
t = (4,5,6)
print("ID is ", id(t))

print(t.__hash__())
print(hash(t))
```

output

```
ID is  26875016
788944837
788944837
-----------------
```

Teach the below after the introduction of class / object / instance / OOPS

```
class A:
    t = (4, 5, 6)
    def __hash__(self,):
        print(A.t.__hash__())

a = A()
x = A()
b = a.__hash__()
# print(b)
print(id(a))
print(id(x))
```

output

788944837
14865288
14865096

========

Tuples are commonly used for small collections of values that will not need to change, such as an IP address and port

```
ip_address = ('10.20.30.40', 8080)
```

The same indexing rules for lists also apply to tuples. Tuples can also be nested and the values can be any valid Python valid values.

A tuple with only one member must be defined (note the comma) this way:

```
one_member_tuple = ('Only member',)
or
one_member_tuple = 'Only member',   # No brackets
```

or just using tuple syntax

```
one_member_tuple = tuple(['Only member'])
```

Declaration of Tuple

Hashing is the process of converting some large amount of data into a much smaller amount (typically a single integer) in a repeatable way so that it can be looked up in a table in constant-time (O(1)), which is important for high-performance algorithms and data structures.

A tuple is an immutable list of values. Tuples are one of Python's simplest and most common collection types, and can be created with the comma operator (value = 1, 2, 3).

Create an empty tuple with parentheses: to = ()
To create a tuple with a single element, you have to include a final comma:
t1 = 'a', // this is tuple
t1 = 'a' // this is string

To create a singleton tuple it is necessary to have a tailing comma.
t2 = ('a',)

Another way to create a tuple is the built-in function tuple

```
t = tuple ('lupins')
print (t)
t = tuple (range(3))
print (t)
```

output

```
('l', 'u', 'p', 'i', 'n', 's')
(0, 1, 2)
```

=======

Tuples are immutable

```
t = (1, 4, 9)
t[0] = 2
```

output

```
  t[0] = 2
TypeError: 'tuple' object does not support item assignment
```

==========

Similarly, tuples don't have .append and .extend methods as list does.

Using += is possible, but it changes the binding of the variable, and not the tuple itself:

```
t1 = (1,2,3)
print(t1)
print(id(t1))
t2 = (10,20,30)
print(t2)
print(id(t2))

print("==========")
```

```
t1 = (t1 + t2)
print(t1)
print(id(t1))
```

output

```
(1, 2, 3)
1632150527168
(10, 20, 30)
1632150369664
============
(1, 2, 3, 10, 20, 30)
1632149909664
```

Notes: initial original tuple values are not modified, where as new elements have been added to NEW tuple

========= ============

Be careful when placing mutable objects, such as lists, inside tuples. This may lead to very confusing outcomes when changing them. For example: Changing item in list is ok, but not in tuple

```
t = [1, 2, 3, [1, 2, 3,]]
print (t[3])
```

```
t[3] = t[3] + [4,5]
print (t[3])
```

outout # this is ok

[1, 2, 3]
[1, 2, 3, 4, 5]
==========

```
t = (1, 2, 3, [1, 2, 3,])
print (t)

t[3] += [4,5] # t[3] = t[3] + [4,5]
print (t)
```

Will both raise an error and change the contents of the list within the tuple:
TypeError: 'tuple' object does not support item assignment

Output

TypeError: 'tuple' object does not support item assignment
(1, 2, 3, [1, 2, 3])

Another code

Be careful when placing mutable objects, such as lists, inside tuples. This may lead to very confusing outcomes when changing them. For example:

Note the output supposed to be ([1, 2, 3,4,5]) // but we get only the original tuple
Of (1, 2, 3, [1, 2, 3,]) // don't use mutable objects inside the tuple
==========

You can use the += operator to "append" to a tuple - this works by creating a new tuple with the new element you "appended" and assign it to its current variable; the old tuple is not changed, but replaced!

This avoids converting to and from a list, but this is slow and is a bad practice, especially if you're going to append multiple times.

```
t = (1, 2, 3, [1, 2, 3,])
print(id(t))
print (t)
print("==========")

a = t+(100,200)
print(a)
print(id(a))
```

output
2210194156184
(1, 2, 3, [1, 2, 3])
(1, 2, 3, [1, 2, 3], 100, 200)
2210197156776

Packing and Unpacking Tuples

```
a = 1, 2, 3 # a is the tuple (1, 2, 3)
```
and
```
a = (1, 2, 3) # a is the tuple (1, 2, 3)
```

are equivalent. The assignment a = 1, 2, 3 is also called packing because it packs values together in a tuple

Note that a one-value tuple is also a tuple. To tell Python that a variable is a tuple and not a single value you can use a trailing comma
```
a = 1 # a is the value 1 (this is int)
a = 1, # a is the tuple (1,) (This is tuple)
```

A comma is needed also if you use parentheses

a = (1,) # a is the tuple (1,)
a = (1) # a is the value 1 and not a tuple
=======

To unpack values from a tuple and do multiple assignments use unpacking AKA multiple assignment

```
x, y, z = (1, 2, 3)
print (x)
print (y)
print (z)
```

output

1
2
3
========

Catch-all variable (see Unpacking Iterables)

In Python 3, a target variable with a * prefix can be used as a catch-all variable

```
first, *more, last = (1, 2, 3, 4, 5, 6)
print (first)
print (*more)
print (last)
```

output

```
1
2 3 4 5
6
```

======

```
*first, more,  = (1, 2, 3, 4, 5, 6)
print (first) # note with out * also it is possible
print (type(first))
print (more)
```

Output

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

What happends if we give * to the first variable

```
*first, more, c = (1, 2, 3, "Kakkan",4, 5, 6,"Anna")
print (*first)
print (more)
print (c)
```

Output
1 2 3 Kakkan 4 5
6
Anna

Another example for catch-all variable

```
a, *b, c = range(5)
print (a)
print (*b)
print (c)
```

output
0
1 2 3
4
==========

Built-in Tuple Functions

Tuples support the following built-in functions

Comparison (the below is for Python 2, so it does not give the expected result
****

If elements are of the same type, python performs the comparison and returns the result.
If elements are different types, it checks whether they are numbers.

1. If numbers, perform comparison.
2. If either element is a number, then the other element is returned.
3. Otherwise, types are sorted alphabetically .

- If we reached the end of one of the lists, the longer list is "larger."
- If both list are same it returns 0.

****

using == operator

```
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 3)
t3 = tuple1 == tuple2
print (t3)
```

output

```
True
--------
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 3, 4)# THIS TUPLE HAS 4 ELEMENTS
t3 = tuple1 == tuple2
print (t3)
```

output

```
False
==========
```

```
tuple1 = ('a', 'b', 'c', 'A', 's', 'd', 'e')
print (len(tuple1))
print ("MAX VALUE", max(tuple1)) #TAKES THE LARGEST ASCII VALUE
print ("MIN VALUE", min(tuple1))# #TAKES THE LARGEST ASCII VALUE

print("==========")
for char in tuple1:
    print(char, "\t", ord(char))
```

output

```
7
MAX VALUE s
```

MIN VALUE A

==========

a 97
b 98
c 99
A 65
s 115
d 100
e 101

---

tuple2 = ('1','2','3')
print (max(tuple2))
output
3

Convert a list into tuple vice versa
lst1= ['a', 'b', 'c', 'A', 's', 'd', 'e','a']
a=tuple(lst1)
print (a)
print (type(a))
output

('a', 'b', 'c', 'A', 's', 'd', 'e', ‘a’)
<class 'tuple'>
========

Tuple concatenation Use + to concatenate two tuples

```
tuple1 =('a', 'b', 'c', 'A', 's', 'd', 'e')
tuple2 = (1, 2, 3, )
tupel3 = tuple1 + tuple2
print (tupel3)
```

output

('a', 'b', 'c', 'A', 's', 'd', 'e', 1, 2, 3)
==========

Tuple Are Element-wise Hashable and Equatable

def hash(__obj: object) -> int
Return the hash value for the given object.

Two objects that compare equal must also have the same hash value, but the reverse is not necessarily true

```
tuple2 = (1, 2, 3, )
print (hash(tuple2))
```

Output

2528502973977326415

Note: this hash value will NOT change even if we modify the object / variable name (ie, tuple2) but if we modify the elements in the tuple, then it CHANGES

```
tuple3 = (1, 2, 3, (5, 6),)
print (hash(tuple3))
```

output

1824638960084655553

Note: Since we have tuple inside tuple, it hashable

___eq___() to check if both hash values are equivalent

```
tuple1 = (1, 2, 3, )
print (hash(tuple1))

tuple2 = (1, 2, 3, )
```

```
print (hash(tuple2))

print(tuple1.__eq__(tuple2))

output
2528502973977326415
2528502973977326415
True
=========
```

Immutable date types inside tuple and how it behaves

```
a= ({'hello'}, [])#, WE PUT A DICT AND SET INSIDE A TUPLE, SET IS MUTABLE, DICT MUTATBLE (IT IS NOT OK)
print (hash(a))

output
    print (hash(a))
TypeError: unhashable type: 'set'
```

Note: Thus a tuple can be put inside a set or as a key in a dict only if each of its elements can.

```
set1 = (10, 20, 30)
set2 =frozenset(set1)

a= (10, 20, set2)
print(a)
print (hash(a))
```

output

5080581608703367281

Note: We declared a set, then it was converted into frozenset, which is immutable. The frozen set was added into a tuple, which is also an immutable (Both are immutable so it is OK)

Indexing Tuples

```
x = (1, 2, 3)
print(x[0])
print(x[1])
print(x[2])
print(x[3])
```

output
print(x[3])
IndexError: tuple index out of range
1
2
3
=========

Indexing with negative numbers will start from the last element as -1:

```
x = (1, 2, 3)
print(x[-1])
print(x[-2])
print(x[-3])
```

output

3
2

1

```
=======
x = (1, 2, 3)
print(x[-4])
  print(x[-4])
IndexError: tuple index out of range
======
x = (1, 2, 3)
print(x[:-1]) # (1, 2)
print(x[-1:]) # (3,) # since -1 is the last value it gives only 3
print(x[1:3]) # (2, 3)
```

output

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

Reversing Elements

```
colors = "red", "green", "blue"
rev = colors[::-1]
print (rev)
# rev: ("blue", "green", "red")
colors = rev
print (colors)
# colors: ("blue", "green", "red")
```

output

```
('blue', 'green', 'red')
('blue', 'green', 'red')
======
```

Or using reversed() (reversed gives an iterable which is converted to a tuple):

```
colors = "red", "green", "blue"
print(reversed(colors))#  """ Return a reverse iterator over the values of the given sequence.
"""
print(tuple(reversed(colors)))
```