Data Structure: tuple:

- A tuple has zero or more elements
- The element of the tuple can be of any type. There is no restriction on the type of the element.
- A tuple   has elements which could be of the same type(homogeneous) or of different types(heterogeneous)
- Each element of a tuple can be referred to by a index or a subscript
- An index is an integer
- Access to an element based on index or position takes the same time no matter where the element is in the tuple – random access
- Tuples are immutable. Once created, we cannot change the number of elements – no append, no insert, no remove, no delete.
- Elements of the tuple cannot be assigned.
- A tuple can not grow and cannot shrink. Size of the tuple can be found using the function len.
- Elements in the tuple can repeat
- Tuple is a sequence
- Tuple is also iterable - is eager and not lazy.
- Tuples can be nested. We can have tuple of tuples.
- Assignment of one tuple to another causes both to refer to the same tuple.
- tuples can be sliced. This creates a new tuple

You may walk through this program to understand the creation and the operations on a tuple.
If the element of a tuple is a list, then the list can be modified by adding elements or removing them. We cannot replace the element of the tuple by assignment even if it is a list.

Creation of tuple:

- empty tuple
    - t1 = ()
    - t2 = tuple()
- two element tuple

    - t3 = (1, 2)
- one element tuple
    - t4 = (10) # NO
    - t5 = (20,) # Yes

There is an ambiguity when we use parentheses around a single expression. Should we consider this as an expression or a tuple? The language considers this as an expression. To make a tuple of single element, we require an extra comma.

```
# name : 1_tuple.py

# tuple
#	is a sequence
#	like a list
#	indexed by int; leftmost element has an index 0
#	select the element using []
#	is immutable
#	once created, cannot be changed
#	length of the tuple cannot change
#	heterogeneous
#	iterable
a = (11, 33, 22, 44, 55)
print(a)
print(a[2]) # 22
print(a[2:4]) # (22, 44)

#a[2] = 222 # NO
#a.append(66) # Error

#ok; a new tuple created
a = a + (111, 222)
print(a)



b = ([12, 23], {34 : 45}, "56" )
```

```
print(b, len(b))

b[0].append(67)  #ok
#b[0] = [78, 89]  #no
#del b[0] # no
#b[0] += [100] # no ; assignment forbidden

a = (11, 33, 22, 44, 55)
for i in a :
        print(i, end = " ")
print()

c = [11, 22, 33, 44]
for i in c :
        print("one")  # 4 times

for i in [c] :
        print("two")  # once

for i in (c) : #  not a tuple
        print("three") # 4 times

for i in (c,) : #  a tuple
        print("four") # once

print( (3, 4) * 2) # (3, 4, 3, 4)
print( (3 + 4) * 2) #  14
print( (3 + 4,) * 2) #  (7, 7)
# tuple of one element requires an extra comma
d = ()
print(d, type(d))

e = (11, 33, 11, 11, 44, 33)
print(e.count(11)) # 3
print(e.count(33)) # 2
print(e.count(55)) # 0
print(e.index(44)) # 4
print(e.index(11)) # 0
print(e.index(55)) # error
```

In this example, we will consider a few uses of tuples.

```
a = 1, 2, 3
```

The above statement creates a tuple of 3 elements. Parentheses are optional.

```
x, y, z = a
```

This will cause the elements of the tuple to be assigned to x, y and z.

We may also use unnamed tuples while assigning to # of variables.

```
a, b = 11, 22
```

The corresponding elements are assigned thereby a becomes 11 and b becomes 22.

```
(a, b) = (b, a) # swaps two variables
```

This causes swapping of two variables. The expressions on the right of assignment are evaluated before the assignment is made. So, even if the variables occur on both sides of assignment operator, there is no ambiguity.

# name: 2_tuple.py

```
a = 1, 2, 3
print(a, type(a))
x, y, z = a
print(x, y, z)
#q, w = a # error; # of variables on the left should match the # of elem in the
tuple

# use of unnamed tuple
a, b = 11, 22
# (a, b) = (11, 22)
print("a : ", a,  " b : ", b)
```

```
# in case of assignment, the right hand side is completely evaluated before
assignment
(a, b) = (b, a) # swaps two variables
# (a, b) = (22, 11)
print("a : ", a,  " b : ", b)

#-----------------

score = { }
#score['gavaskar'] = 10000
# many times, key has components : composite key
# should be immutable; cannot be a list
#score[['sunil', 'gavaskar']] = 10000 # NO
#score[['rohan', 'gavaskar']] = 1000 # NO

score[('sunil', 'gavaskar')] = 10000
score[('rohan', 'gavaskar')] = 1000
print(score)
print(score['rohan', 'gavaskar'])
```

Tuples are also used as keys of dict whenever the key has multiple components – key of the dict is a composite. The key of a dict should be immutable. So the key cannot be a list, but it can be a tuple.

Data structure: str : string

A string is a sequence of characters. Python directly supports a str type; but there is no character type.

- A string has zero or more characters
- Each character of a string can be referred to by a index or a subscript
- An index is an integer
- Access to an element based on index or position takes the same time no matter where the element is in the string – random access
- strings are immutable. Once created, we cannot change the number of elements – no append, no insert, no remove, no delete.
- Elements of the string cannot be assigned.

- A string can not grow and cannot shrink. Size of the string can be found using the function len.
- strng is a sequence
- string is also iterable - is eager and not lazy.
- Strings cannot be nested.
- strings can be sliced. This creates a new string.

There are 4 types of string literals or constants.
a) single quoted strings
b) double quoted strings
There is no difference between the two. In both these strings, escape sequences like \t, \n are expanded.
We can use double quotes in a single quoted string and single quote in double quoted string without escaping.
These strings can span just a line – cannot span multiple lines.

c) triple quoted strings
We can create a string spanning multiple lines by using either three single quotes or three double quotes as delimiters.
These strings are also used for documentation.
__doc__ : document for the whole program. It stores the string literal specified in the beginning of the program.
We can __doc__ string for a class and for a function.

d) raw strings
There are cases where the escape sequence should not be expanded – we require such strings as patterns in pattern matching using regular expressions. In such cases, we prefix r to the string literal – it becomes a raw string.

```
# name : str3.py

"""
this program
is about playing
with strings
```

```
"""
#  stored in a variable : __doc__
# strings
#       sequence
#       indexed
#       leftmost index : 0
#       immutable
#       no character type
#       can be sliced
#       cannot assign

a = "rose"
print(a, type(a), a[2], type(a[2]))
#a[0] = 'b' # error

# make a string:
# 1. single quotes
# 2. double quotes
#       no difference between them
#       escape sequences are expanded

s1 = "this is a \n string"
s2 = 'this is a \n string'
print(s1, type(s1))
print(s2, type(s2))

x = "indira gandhi is nehru's daughter's name"
print(x)

# 3. raw string
# no escaping
s3 = r"this is a \n string"
print(s3, type(s3))

# 4. triple quoted string
```

```
s4 = """
we love python
very much
"""
print(s4, type(s4))
print("document string : ", __doc__)
```

The following example shows how to play with the strings. The str type has lots of useful functions.

- Build a string in stages
  ss = “” # create an empty string
  ss = ss + “something” # create a new string by concatenation
- call some utility function like upper or replace
  x = “abcd”; x.upper(); print(x)
  We will observe that x has not changed. Remember that x is immutable. In case we want the original string to change, assign the result of the function call back to the same variable – thus recreating the variable.
  x = “abcd”; **x = x.upper()**; print(x)
- Observe that replace will return a new string modifying every occurrences of the old string with the replace string. We can control the number of changes by using a count as the third argument.
- Index finds the leftmost occurrence of a substring in the given string. Finding the nth occurrence or replacing the nth occurrence become programming exercises.

```
# name : 4_str.py
"""
s = 'mohanDas Karamchand gandhi'
# print "m K gandhi"
# make a list of words by splitting
# output the first char of each word but for the last; output the last word
ss = ''
namelist = s.split()
for name in namelist[:len(namelist)-1]:
        #print(name[0])
        ss = ss + name[0] + " "
#print(namelist[-1]) # last elem
ss = ss + namelist[-1]
print(ss)
#ss.upper()  # NO; does not change the str; returns a new changed string
ss = ss.upper()
print(ss)

ss = ss.title()
print(ss)

s  = s.title()
print(s)
"""
mylist = [
        "indira gandhi",
        "m k gandhi",
        "rahul gandhi",
        "jawaharlal nehru",
        "sardar patel",
        "brijesh patel"
]
for w in mylist :
        if w.endswith('gandhi') :
                print(w)

s = "bad python bad teacher bad lecture"
print(s.replace('bad', 'good')) # default : all occurrences
print(s.replace('bad', 'good', 1))

s = "bad python bad teacher bad lecture"

# find the leftmost bad
print(s.index('bad'))
```

```
# find the second bad from left
print(s.index('bad', s.index('bad') + len('bad')))

i = s.index('bad', s.index('bad') + len('bad'))
print(s[i:].replace('bad', 'worst', 1))
print(s[:i] + s[i:].replace('bad', 'worst', 1))
```

Some encoding these are!

In the first case, the first letter of each word is printed at the end.

In the second case, after each character, a p is printed.

```
# name: 5_str.py
s = "we love python very much"
for w in s.split():
        print(w[1:], end = "")
        print(w[0], end = " ")
print()

for ch in s:
        print(ch, end = "")
        print('p', end = "")
print()
```

Outputs are:

```
ew ovel ythonp eryv uchm
wpep plpopvpep pppyptphpopnp pvpeprpyp pmpupcphp
```