MCT-242 : COMPUTER PROGRAMMING-I

using Python 3.9

Prepared By:

Mr. Muhammad Ahsan Naeem

YouTube Playlist

https://www.youtube.com/playlist?list=PLWF9TXck7O_wMDB-VriREZ6EvwkWLNB7q

LAB 20: TUPLE: CLO-4

Like List, we have **Tuple** as sequence data type in Python. By sequence it means that Tuple can hold multiple objects and that too in an order.

The syntax to create a Tuple is specifying the objects inside the parentheses **()** rather than the square brackets **[]** in case of a list. See here how a Tuple named as a is defined:

```
a=(2,3,6,False,'Lecture')
```

And if we print it as:

```
print(a)
```

It will be displayed as:

```
(2, 3, 6, False, 'Lecture')
```

A comparison of List and Tuple operations are given in Table o next page. If you see that, all operations are exactly same for List and Tuple. The built-in Python function **sorted** returns a sorted list when applied on a list but in case it is applied on a Tuple, instead of returning a Tuple, it returns a List.

Of course, the first question arises is why we have Tuple when we can do the same thing with List. We will come to the differences between List and Tuple but at first, even with the same operations of the above table, there can be difference in performance. The two key performance indicators for a program are:

i) The memory consumption
ii) Time of execution

```python
# List
a=[2,3,6,False,'Lecture']
print(a)    # [2, 3, 6, False, 'Lecture']
## Elements can be accessed using Index
print(a[3])     # False
print(a[-1])    # Lecture
print(a[1:4])   # [3, 6, False]
## Membership can be checked using in operator
print(3 in a)   # True
## Create empty List
x=list() #or x=[]
## List is Iterable
for i in a:
    print(i)    # All elements will be printed one-
by-one

## List Operations
b=[2,4,1]
c=a+b   # [2, 4, 6, False, 'Lecture', 2, 3, 1]
print(c)
print(b*2)  #[2, 3, 1, 2, 4, 1]
## Comparison Operators
print(a==b) #False
print(a>b)  #False (because 1st element is same.
            # and second element of b is greater
            # than second element of a)

## Built-in Python Functions for List
print(len(b))   # 3
print(max(b))   # 4
print(min(b))   # 1
print(sum(b))   # 7
print(all(b))       # True
print(any(b))       # False
print(sorted(b))    # [1, 2, 4]
```

```python
# Tuple
a=(2,3,6,False,'Lecture')
print(a)    # (2, 3, 6, False, 'Lecture')
## Elements can be accessed using Index
print(a[3])     # False
print(a[-1])    # Lecture
print(a[1:4])   # (3, 6, False)
## Membership can be checked using in operator
print(3 in a)   # True
## Create empty Tuple
x=tuple() #or x=()
## Tuple is Iterable
for i in a:
    print(i)    # All elements will be printed one-
by-one

## Tuple Operations
b=(2,4,1)
c=a+b   # (2, 4, 6, False, 'Lecture', 2, 3, 1)
print(c)
print(b*2)  #(2, 3, 1, 2, 4, 1)
## Comparison Operators
print(a==b) #False
print(a>b)  #False (because 1st element is same.
            # and second element of b is greater
            # than second element of a)

## Built-in Python Functions for Tuple
print(len(b))   # 3
print(max(b))   # 4
print(min(b))   # 1
print(sum(b))   # 7
print(all(b))       # True
print(any(b))       # False
print(sorted(b))    # [1, 2, 4]
```

We can check the memory size taken by some variable (in Bytes) using the `getsizeof` method available in `sys` module. Here is comparison of one List and one Tuple with exactly same elements:

```
from sys import getsizeof
a=[4,5,9.2,'Computer',True]
b=(4,5,9.2,'Computer',True)
print(f'Size of list: {getsizeof(a)}')
print(f'Size of tuple: {getsizeof(b)}')
```

And the output is:

```
Size of list: 120
Size of tuple: 80
```

So, Tuple takes lesser memory!

We can find the execution time of a code using the `timeit` method available inside the `timeit` module. In this function we need to specify a statement whose time of execution we want to calculate and number of times we want to execute that statement. Since the time of execution of a simple list creation or a tuple creation is extremely low, so for better comparison we will run that statement 100 Million times!

Here is the code:

```
from timeit import timeit

t1=timeit(stmt='a=[1,2,3,4,5]',number=100_000_000)
t2=timeit(stmt='a=(1,2,3,4,5)',number=100_000_000)

print(f'List time test:{t1}')
print(f'Tuple time test:{t2}')
```

The output of this code will vary from each time we run the code. Because the execution time depends on a lot of factors and each time those factors might be different (for example available free RAM at the time of execution). In my case, this is the output:

```
List time test:7.9229881
Tuple time test:2.3392847999999997
```

So, Tuple is faster than List!

In both criteria i.e., Memory Consumption and Execution Time, Tuple is more efficient. So, does that mean we should always use Tuple instead of List?

No, because that efficient behavior of a Tuple over a List, comes with a price. Let us display the List class methods and the Tuple class methods separated by three blank lines:

```
print(dir(list))
print()
print()
print()
print(dir(tuple))
```

This is the output:

```
['__add__', '__class__', '__class_getitem__', '__contains__',
'__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']



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

Ignore the methods written in between double underscore since those are **Special Methods** also known as **Magic Methods**, used internally for basic List or Tuple operations. Other than those, there are 11 methods in list class starting from append to sort. Whereas there are only two in Tuple class; **`count`** and **`index`**. The working of **`count`** and **`index`** is exactly same as of the List class, i.e. **`count`** will return the number of times an element (provided as input argument) is present inside the Tuple and **`index`** will return the index of the element (provided as input argument) inside the Tuple.

So, it means, we do not get flexibility of different operations in Tuple. Other than that, there are few other differences between List and Tuple which are discussed below.

Tuple is Immutable while List is Mutable:

In case of a list, we can change any element of it as shown here:

```
a=[2,3,10]
a[0]=5
print(a)     # Will print [5, 3, 10]
```

But in case of Tuple, if we try to do this as:

```
a=(2,3,10)
a[0]=5
```

This will generate the following error:

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

Tuple is **Immutable** data types while the list is **Mutable**. In simple words, a list when created can be changed, but a Tuple cannot be changed once it is created. There is a lot more on Mutability and Immutability of a data type and we will see that detail in one future lab manual.

Packing and Unpacking the Tuple:

We create a list like this:

```
t=(3,4,5)
```

But there we can remove the paratheses and still it will be a Tuple as:

```
t=3,4,5
```

This is known as **Packing** the objects into a Tuple.

Like **Packing**, we can also **Unpack** a Tuple.

We can retrieve all elements of the tuple as:

```
a,b,c=t
```

You can print the values; **a** will be **3**, **b** will be **4** and **c** will be **5**. We cannot do it on lesser elements, for example

```
a,b=t
```

This will generate an error:

```
ValueError: too many values to unpack (expected 2)
```

And if we do it for single element as:

```
a=t
```

This is not Unpacking. We are having another variable a as Tuple same as the Tuple t.

How ever if we do something like:

```
t=(3,4,5,6,7,8,9)
a,b,*c=t
```

This will not generate error. The first element of the Tuple will get unpacked to **a**, second will get unpacked to **b** and remaining elements will be assigned to **c** as a list. We can place the ***** with any of the variables on the left side but not with more than one.

If we will be doing like this:

```
t,u,v=3,4,5
```

The variable **t**, **u** and **v** are not Tuples but integers.

We studied that Python support multiple assignment as being done in **t,u,v=3,4,5** but internally the right side expression i.e., **3,4,5** is converted to a Tuple (Packed) and then unpacked to assign the values to variables **t**, **u** and **v**.

You can consider the above statement as:

```
x=3,4,5 # Packing into Tuple
t,u,v=x # Unpacking the Tuple
```

The variable **x** is of Tuple Data Type. So, it is because of Tuple, the multiple assignment works in Python.

In fact, these four statements are equivalent:

```
m,n = 10,20
m,n = (10,20)
(m,n) = 10,20
(m,n) = (10,20)
```

And this is exactly why we can very easily swap the values between two or more variables as:

```
m,n = n,m
```

Now let's see a simple function having two output arguments:

```
def test(a,b):
    x=a+b
    y=a-b
    return x,y
```

The function is simply returning the sum and the difference of the two input arguments. And when we use it, we provide two variables to store those two outputs as:

```
m,n=test(10,2)
print(m)     # prints 12
print(n)     # prints 8
```

We generally say that the function is returning two values, but in reality, it is returning just the one, which is a Tuple with two elements. Let's see that over here:

```
p=test(10,2)
print(p)        # prints (12, 8)
print(type(p))  # prints <class 'tuple'>
```

So basically when a function returns multiple values like return **x**,**y** it actually packs those to a Tuple and returns a single Tuple. But when we use that as:

```
m,n=test(10,2)
```

There is actually a Tuple on the right side which gets unpacked and values are assigned to **m** and **n**.

Singleton Tuple:

Let's define following Tuples and print their values and datatypes:

```
t1=()
print(f't1={t1}')
print(type(t1))
print()
t2=(1)
print(f't2={t2}')
print(type(t2))
print()
t3=(1,2)
print(f't3={t3}')
print(type(t3))
print()
t4=(1,2,3)
print(f't4={t4}')
print(type(t4))
```

This is the output:

```
t1=()
<class 'tuple'>

t2=1
<class 'int'>

t3=(1, 2)
<class 'tuple'>

t4=(1, 2, 3)
<class 'tuple'>
```

Everything looks perfect except for the second Tuple which is printed as `Integer` class object. This is because for a Tuple with single value we have to put a `,` after that element as:

```
t2=(1,)
print(f't2={t2}')
print(type(t2))
```

And now the output will be:

```
t2=(1,)
<class 'tuple'>
```

Example Use Cases:

Now let's see some cases where we can use the Tuples.

Scenario 1) Here is the list of zodiac signs and date ranges against each of them:

Capricorn →	December 22 to January 19
Aquarius →	January 20 to February 18
Pisces →	February 19 to March 20
Aries →	March 21 to April 19
Taurus →	April 20 to May 20
Gemini →	May 21 to June 20
Cancer →	June 21 to July 22
Leo →	July 23 to August 22
Virgo →	August 23 to September 22
Libra →	September 23 to October 22
Scorpio →	October 23 to November 21
Sagittarius →	November 22 to December 21

We can write a program to display the zodiac sign against entered day and month using `if-else` statements. But a better approach can be using a nested Tuple like this:

```
def zodiac(day,month):
    signs=(
        ('Capricorn',19),('Aquarius',18),('Pisces',20),
        ('Aries',19),('Taurus',20),('Gemini',20),
        ('Cancer',22),('Leo',22),('Virgo',22),
        ('Libra',22) ,('Scorpio',21),('Sagittarius',21),
          ('Capricorn',))
    if(day<=signs[month-1][1]):
        return signs[month-1][0]
    else:
        return signs[month][0]

## Main Program ##
print(zodiac(15,7)) #prints Cancer
print(zodiac(25,7)) #prints Leo
```

The information of zodiac sign with the respective day for each month is one inner Tuple and all Tuples are stored inside the outer Tuple.

This could also be done using the Lists instead of Tuples as:

```
def zodiac(day,month):
    signs=[
        ['Capricorn',19],['Aquarius',18],['Pisces',20],
        ['Aries',19],['Taurus',20],['Gemini',20],
        ['Cancer',22],['Leo',22],['Virgo',22],
        ['Libra',22],['Scorpio',21],['Sagittarius',21],
        ['Capricorn']]
    if(day<=signs[month-1][1]):
        return signs[month-1][0]
    else:
        return signs[month][0]

## Main Program ##
print(zodiac(15,7)) #prints Cancer
print(zodiac(25,7)) #prints Leo
```

Or we could do it as a List of inner Tuples as:

```
def zodiac(day,month):
    signs=[
        ('Capricorn',19),('Aquarius',18),('Pisces',20),
        ('Aries',19),('Taurus',20),('Gemini',20),
        ('Cancer',22),('Leo',22),('Virgo',22),
        ('Libra',22) ,('Scorpio',21),('Sagittarius',21),
        ('Capricorn',)]
    if(day<=signs[month-1][1]):
        return signs[month-1][0]
    else:
        return signs[month][0]

## Main Program ##
```

```
print(zodiac(15,7)) #prints Cancer
print(zodiac(25,7)) #prints Leo
```

Scenario 2) If we have to store the properties of one element of the periodic table, then Tuple is most suited for that because those properties will never change. Here the first four elements of the periodic table are defined with the properties name, symbol, atomic number and atomic mass:

```
e1=('Hydrogen','H',1,1)
e2=('Helium','He',2,4)
e3=('Lithium','Li',3,7)
e4=('Beryllium','Be',4,9)
```

We can define all elements like that and then can place all of those inside a List representing the **Periodic Table**. We can also place these inside the nested list to get the Period Number and the Group Number of the elements from the list indices. Here just the four created elements are added into a list:

```
perTable=[e1,e2,e3,e4]
print(perTable)
```

The output is:

```
[('Hydrogen', 'H', 1, 1), ('Helium', 'He', 2, 4), ('Lithium', 'Li',
3, 7), ('Beryllium', 'Be', 4, 9)]
```

List is the most used sequence because we get a variety of list class methods for different queries/operations on list objects. In above case, we are preferring to use a Tuple for element since element properties will never change. Then those are placed inside the list so that we can apply different queries on the list of elements.

For example, let's print just the **symbols** of the elements whose **atomic mass** is between **4** and **8**. We can loop over the list and the loop variable e.g. **`element`** will be the Tuple inside the list. Then inside **`for`** loop, we can unpack the tuple to get the element properties and use the appropriate condition.

```
for element in perTable:
    name,symbol,no,mass=element
    if(4<=mass<=8):
        print(symbol)
```

We can unpack the loop variable **`element`** directly there at the **`for`** loop statement as shown here:

```
for name,symbol,no,mass in perTable:
    if(4<=mass<=8):
        print(symbol)
```

Scenario 3) A color in digital world is represented as a **RGB code**; where R is for Red, G is for Green and B is for Blue. These red, green and blue colors are known as the **Primary Colors** and all other colors can be derived by mixing these colors. In digital representation of the colors, the three R, G and B values are represented in 1-Byte (8-Bits) and hence in decimal their values are from 0-255.

Hence **R,G,B=255,0,0** means a pure red color and so on different RGB codes for different colors.

In different graphical applications, as a developer, we need to specify the colors as RGB codes. For one application we can define all the colors in the program and a tuple a best suited for this.

So, we can define color as:

```
color1=(100,200,100)
```

For one graphical application we can save all selected colors in a list, for example as:

```
color1=(100,200,100)
color2=(20,80,120)
color3=(60,100,200)
color4=(20,0,140)
allColors=[color1,color2,color3,color4]
print(allColors)
```

Let's say we want to print all colors inside the list, one on each line. We can simply do it as:

```
for c in allColors:
    print(c)
```

The loop variable **c** is the list element which is a Tuple representing a color.

Now suppose we want to display only those colors whose R, G and B values all are different from each other. For that we can unpack **c** inside the loop and apply the conditions as:

```
color1=(100,200,100)
color2=(20,80,120)
color3=(60,100,200)
color4=(20,0,140)
allColors=[color1,color2,color3,color4]
for r,g,b in allColors:
    if(r!=g and g!=b and b!=r):
        print((r,g,b))
```

The output will be:

```
(20, 80, 120)
(60, 100, 200)
(20, 0, 140)
```

Scenario 4) A cartesian point on **XY Plane** will have two components: **x-component** and **y-component**. One such point can be represented as Tuple with two values. And then if we want to represent a **Polygon** with **n** vertices, it can be a list of **n** Tuples.

Tasks:

[1] Considering the Scenario-4 of **XY** Points on XY Plane, a polygon will be a list of such points. Define a function named as **`polyPerimeter`** which will take a polygon (list of points as Tuple) as input argument and will return the perimeter of that Polygon. You can create more function if you want.

[2] Consider the case that a student information is defined as a tuple in this format (First Name, Last Name, Reg. No, List of courses registered by him/her).
Four examples of students in this format are shown here and finally both are added into a list.

```
s1=('Ahmad','Anwar','MCT-UET-01',['CP','LA'])
s2=('Ali','Jamal','MCT-UET-02',['LA'])
s3=('Muneeb','Akhtar','MCT-UET-03',['Phy','CP','LA'])
s4=('Rizwan','Khan','MCT-UET-04',['Statistics','Phy'])
```

You can add a few more students inside the list and then your program should display:

- The registration numbers of all students who are registered in the course CP.
- The full names of the students who have registered exactly 3 courses.