List, Set, Tuple ,Dictionaries, Data type conversion

L6 and L7

List

- List is a data type of Python used to store multiple values of different types of data at a time. List are represented with [].

- A list can be created by putting comma separated values between square brackets [].

The following program shows creation of two lists namely list1 and list2 :

```
list1 = [1, 2, "one", "hi"]
list2 = [4, 5, "hello"]
```

- Values stored in the list are accessed using an index.

- Index range between 0 to n-1, where n is the number of values in the list

- Python allows negative indexing for lists. The index of -1 refers to the last value of the list, -2 refers to the second last value of the list and so on.

Printing Lists

The lists can be printed using the in-built function **print()** as shown below.

```
list1 = [1, 2, "one", "hi"]
print(list1) # will print output as follows

Output :
[1, 2, 'one', 'hi']
```

```
list2 = [4, 5, "hello"]
print(list2) # will print output as follows

Output :
[4, 5, 'hello']
```

Accessing Values in Lists: A value at a particular index in a list is accessed using listname[index]

```
list2 = [4, 5, "hello"]
print(list2[1]) # prints the value present at 1st index in list2

Output :
5
```

Slicing lists: Slicing is used to access a subset of a list. For a list l = [1, 4, 5, 7, 4, 15], a subset of list from 2nd index to 4th index is obtained by l[2:4] which is equal to [5,7]. Slicing can be understood using the following examples:

```
Let us assume that we have a list l = [1, 4, 6, 22, 44, 12, 55, 66]
Slicing expression          Value                   Explanation
l[1:4]                      [4, 6, 22]              The starting index is 1 and ending index is 4. (Observe
that value at the index 4 is not included in the result)
l[3:-3]                     [22, 44]                The starting index is 3 and ending index is -3 which
represents third last element i.e 12
l[3:]                       [22, 44, 12, 55, 66]    The stopping index is not specified so slicing is done
till the end of the list.
l[:4]                       [1, 4, 6, 22]           The starting index is not given so 0 is considered.
```

Concatenating lists: We can concatenate two lists using (+) operator.

```
list1 = [1, 2, 'one', 'hi']
list2 = [4, 5, 'hello']
print(list1 + list2) # will print output as follows

Output:
[1, 2, 'one', 'hi', 4, 5, 'hello']
```

Repeating lists: We can print a list multiple times using (*) operator as shown below:

```
list2 = [4, 5, 'hello']
print(2 * list2) # will print output as follows

Output :
[4, 5, 'hello', 4, 5, 'hello']
```

Working with nested lists: The items of the list can be lists themselves, which means that the lists can be nested.

Let us consider example:

```
list1 = [23, 5.65, ["A", 34.23], "India"]
```

In the above example, the third element (i.e **index = 2**) of list1 is a `list`.

We access the **3rd item** by list1[2], which is a list and the second item of this list can be accessed using `index 1`. So, **list1[2] [1]** will be `34.23`.

Sample Question:

Write the missing code below to understand **List Concatenation**. Follow the instructions given as comment lines in the program.

Sample Input and Output:

```
List1 Elements are: [1.0, 2.3, 'hello']
List2 Elements are: ['hi', 8.3, 9.6, 'how']
List after Concatenation: [1.0, 2.3, 'hello', 'hi', 8.3, 9.6, 'how']
```

Solution:

```
list1 = [1.0, 2.3, "hello"]
list2 = ["hi", 8.3, 9.6, "how"]
print("List1 Elements are:",list1)
print("List2 Elements are:",list2)
print("List after Concatenation:",list1+list2)
```

Sets

- A set is a mutable data type that contains an unordered collection of items.

- Every element in the set should be unique (no duplicates) and must be immutable (which cannot be changed). But the set itself is mutable. We can add or remove items / elements from it.

Note : Mutable data types like list, set and dictionary cannot become elements of a set.

- The set itself is mutable i.e. we can add or remove elements from the set.

The main uses of sets are:

- Membership testing
- Removing duplicates from a sequence
- Performing mathematical operations such as intersection, union, difference, and symmetric difference
- A set is represented with { }.

Creation of sets

A **set** is created by placing all the items (elements) inside curly braces **{}**, separated by comma or by using the built-in function set().

Empty set

An empty set can be created using built-in **set()** function.

```
myset = set()
print(myset) # will print empty set.
print(type(myset)) # will print type of myset.

Output :
set()
<class 'set'>
```

Another way to create a set is to put all elements inside curly braces separated by commas.

```
myset1 = {1, 2, 3}
print(myset1) # will print the elements of a set.
print(type(myset1)) # will print type of myset1.

Output :
{1, 2, 3}
<class 'set'>
```

Note : Empty curly braces { } does not make an empty set in Python, it makes an empty dictionary instead. Dictionary data type is introduced in the upcoming lessons.

```
test = { }
print(type(test))

Output :
<class 'dict'>
```

Which of the following options are correct?

1. A set is an ordered collection of unique items.
2. myset = { }, creates an empty set.
3. A set is represented using curly braces { } .
4. Set allows duplicate elements.
5. Set is a immutable data type.
6. The elements of a set are mutable.

A. 1,2,3 and 4
B. 2,3 and 4
C. 3 and 4
D. Only 3

Which of the following options are correct?

1. A set is an ordered collection of unique items.
2. myset = { }, creates an empty set.
3. A set is represented using curly braces { } .
4. Set allows duplicate elements.
5. Set is a immutable data type.
6. The elements of a set are mutable.

A. 1,2,3 and 4
B. 2,3 and 4
C. 3 and 4
D. Only 3

Tuples

- A tuple is a data type similar to list.
- The major differences between the two are: Lists are enclosed in square brackets [] and their elements and size can be changed (mutable), while tuples are enclosed in parentheses () and their elements cannot be changed (immutable).
- Tuples can be thought of as read-only lists.

Note: Since a tuple is immutable, iterating through tuple is faster than with list. This gives a slight **performance improvement.**

- Once a tuple is defined, we cannot add elements in it or remove elements from it.
- A tuple can be converted into a list so that elements can be modified and converted back to a tuple. Conversion of a tuple into a list and a list into a tuple is discussed in the later sessions.
- A tuple is represented using parenthesis ( ).

Creation of a Tuple:

Tuples can be created using built-in function called `tuple()`.

An empty tuple can be created using `tuple()` function as follows :

```
tuple1 = tuple() # Creating an empty tuple using tuple() function
print(tuple1) # Printing tuple1

Output :
()
```

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

```
mytuple = (1, 2, 3, "Data types") # mytuple = 1, 2, 3, "Data types" will also work.
print(mytuple)
print(type(mytuple))

Output :
(1, 2, 3, "Data types")
<class 'tuple'>
```

```
tuple2 = (1, 2, 3, "data types") # creating tuple with different type of values
print(tuple2) # print created tuple
(1, 2, 3, 'data types')
```

```
print(type(tuple2)) # checking type of object
<class 'tuple'>
```

Care should be taken when a tuple with a single element is to be created.

Consider the following program :

```
mytuple = (1)
print(mytuple) # Will print integer 1 instead of a tuple.
print(type(mytuple)) # will print output as follows

Output :
1
<class 'int'>
```

Closely observe the output to identify that the data type of mytuple is an integer but not a tuple.

One-element tuples look like:

```
(1,)
```

or

```
1,
```

Note: The trailing comma is mandatory for the one-element tuples.

One-element tuples can be created as follows :

```
mytuple = (1,)
print(mytuple) # Will print tuple (1,)
print(type(mytuple)) # will print output as follows

Output :
(1,)
<class 'tuple'>
```

Multiple-element tuple looks like :

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

or

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

Note : The trailing comma is completely optional for the multiple-element tuples.

Select all the correct statements given below.

Which of the following options are correct?

1.Tuples are used to store similar type of data.

2.tuple1 = (1.0) is correct way to create a tuple with single element.

3.tuples are immutable.

4.Lists are immutable.

5.Converting a tuple into a list and list into tuple is possible.

6.Lists are faster than tuples.

A. 1,2,3
B. 2,3
C. 3,5
D. 2,3,4

Which of the following options are correct?

1.Tuples are used to store similar type of data.

2.tuple1 = (1.0) is correct way to create a tuple with single element.

3.tuples are immutable.

4.Lists are immutable.

5.Converting a tuple into a list and list into tuple is possible.

6.Lists are faster than tuples.

A. 1,2,3

B. 2,3

C. 3,5

D. 2,3,4

Dictionaries

Dictionary is an **unordered** collection of key and value pairs.

> **Note** : While other compound data types (like lists, tuples and sets) have only value as an element, a dictionary has a `key` : `value` pair

General usage of dictionaries is to store key-value pairs like :

- Employees and their wages
- Countries and their capitals
- Commodities and their prices

➢ In a dictionary, the keys should be unique, but the values can change. For example, the price of a commodity may change over time, but its name will not change.

➢ Immutable data types like number, string, tuple etc. are used for the key and any data type is used for the value.

➢ Dictionaries are optimized to retrieve values when the keys are known.

➢ Dictionaries are represented using key-value pairs separated by commas inside curly braces {}. The key-value pairs are represented as key : value. For example, daily temperatures in major cities are mapped into a dictionary as { "Hyderabad" : 27 , "Chennai" : 32 , "Mumbai" : 40 }.

Creating a dictionary:

A dictionary can be created in two ways.

- Using the built-in dict() function.
- Assigning elements directly.

1. Using built-in `dict()` function.

An empty dictionary can be created as follows :

```
mydict = dict() # Creating an empty dictionary called mydict
print(type(mydict)) # Printing data type of mydict.
print(mydict)  # Prints empty dictionary.

Output :
<class 'dict'>
{}
```

A dictionary with elements can be created as follows :

```
mydict = dict(Hyderabad = 20, Delhi = 30) # A dictionary with two key pairs is created.
print(mydict) # Prints the dictionary

Output :
{'Hyderabad': 20, 'Delhi': 30}
```

Note : The two key pairs are specified in the dict() function as comma separated `key` = `value`.

2. Assigning elements directly.

A dictionary is created using direct assignment as follows :

```
mydict = {1:"one", 2 :"two", 3:"three"}  # Create a dictionary with three key-value pairs.
print(mydict) # Printing the dictionary
print(type(mydict)) # will print output as follows

Output :
{1:"one", 2 :"two", 3:"three"}
<class 'dict'>
```

Which of the following are the correct options?

1.Dictionary is a Python data type to store multiple values.

2.We use parenthesis () to define a dictionary.

3.In dictionary we represent an element in the {key-value} format.

4.Keys of the dictionary cannot be changed.

5.dictionary() function is used to create an empty dictionary.

A. 1,2,3

B. 1,4

C. 1,4,5

D. 1,3,5

Which of the following are the correct options?

1.Dictionary is a Python data type to store multiple values.

2.We use parenthesis () to define a dictionary.

3.In dictionary we represent an element in the {key-value} format.

4.Keys of the dictionary cannot be changed.

5.dictionary() function is used to create an empty dictionary.

A. 1,2,3

B. 1,4

C. 1,4,5

D. 1,3,5

Accessing elements of Dictionary

We cannot use numerical index (as in lists, tuples and strings) to access the items/elements of the dictionaries as dictionaries are unordered.

(Imagine all the items of a dictionary are put in a bag and jumbled, so there is no order and we cannot retrieve the items using a sequential index.)

The elements of the dictionary can be **retrieved/accessed** in 2 ways.

1. Using the keys of the dictionary.
2. Using the `get()` method.

1. Using the keys of the dictionary.

```
capitals = {"U.S.A" : "Washington D.C", "India" : "New Delhi", "Nepal" : "Kathmandu"} # Creating a dictionary
print(capitals["India"]) # Printing the value with the key "India" i.e. "New Delhi".
print(capitals["Nepal"]) # Printing the value with the key "Nepal" i.e. "Kathmandu".

Output :
New Delhi
Kathmandu
```

Trying to access an element in the dictionary using a key that is not present in the dictionary, results in a `Key Error`.

```
capitals = {"U.S.A" : "Washington D.C", "India" : "New Delhi", "Nepal" : "Kathmandu"} # Creating a dictionary,
print(capitals["India"]) # Printing the value with the key "India" i.e. "New Delhi".
print(capitals["Australia"]) # Since the key "Australia" is not present, it results in a Key Error

Output :
New Delhi
Traceback (most recent call last):
    File "1.py", line 3, in
        print(capitals["Australia"])
KeyError: 'Australia'
```

2. Using the `get()` method.

```
capitals = {"U.S.A" : "Washington D.C", "India" : "New Delhi", "Nepal" : "Kathmandu"} # Creating a dictionary
print(capitals.get("India")) # Printing the value with the key "India" i.e. "New Delhi".
print(capitals.get("Nepal")) # Printing the value with the key "Nepal" i.e. "Kathmandu".

Output :
New Delhi
Kathmandu
```

Trying to access a key that is not present in the dictionary using the get() method results in `None`

```
capitals = {"U.S.A" : "Washington D.C", "India" : "New Delhi", "Nepal" : "Kathmandu"} # Creating a dictionary
print(capitals.get("India")) # Printing the value with the key "India" i.e. "New Delhi".
print(capitals.get("Australia")) # Since the key "Australia" is not present, it results in a None
Output :
New Delhi
None
```

Write the missing code in the below program to retrieve the elements of the dictionary **using keys**.

More points in relation to Dictionary

Let us create a dictionary called **dict1** with three **keys** named as `name`, `number` and `age`, **values** of dictionary are `Jay` , `514`, and `12` and then try to get values using respective keys.

1. **Creating a dictionary:**

```
dict1 = {"name":"Jay", "number":514, "age":12}
print(dict1) # will print output as follows
{'name': 'Jay', 'number': 514, 'age': 12}
```

2. **Let us retrieve the values using their keys as index:**

```
print(dict1['age']) # using key called 'age', we can get value 12
12  # value of respective key 'age'
print(dict1['number']) # using 'number' as key to get value 514
514
```

3. **Let us find what happens when we try to retrieve an element (key) which is not present in the dictionary:**

```
print(dict1['place'])  # using key called 'place' we are trying to get value of place but doesn't exist in
Dict.
Traceback (most recent call last):   # so it returns an error as "KeyError"
 File "<stdin>", line 1, in <module>
KeyError: 'place'
```

If the **key** is not there in the dictionary, we get a `KeyError` .

4. **We can change (update) the values in a dictionary.**

```
dict1['name'] = "Krithika"
print(dict1) # will print output as follows
{'name':'Krithika' ,'number':514, 'age':12}
```

Data Type Conversions using conversion functions

1. **int(x, base)**: This function converts **x** to an integer of specified base. If base is not specified, it defaults to **10**.

The syntax of `int()` function is:

```
int(x=0, base=10)

    • x - Number or string to be converted to integer. Default argument is zero.
    • base - Base of the number in x. Can be 0 (code literal) or 2-36.
```

Consider the following program to understand the working of `int()` function.

```
s = "0011" # A binary string.
print(int(s, 2)) # Converts string type to int type using int() with base 2
print(int(s)) # base not specified, it defaults to 10

Output:
3
11
```

If "0011" is considered with base 2, then its integer value will be **3**, whereas if "0011" is considered with base 10, then its integer value will be **11**.

2. **float(x)**: This function is used to convert any data type to a floating point number. The float() function returns a floating point number from a number or a string.

The syntax of `float()` function is:

```
float(x)

    • x (Optional) - number or string that needs to be converted to floating point number. If it's a string,
      the string should contain decimal points
```

Usage	Output	Explanation
`print(float(23.4))`	`23.4`	The parameter is already a float number. So no conversion happens
`print(float(9))`	`9.0`	The integer `9` is converted to float `9.0`
`print(float("32"))`	`32.0`	The string `"32"` is converted to float `32.0`
`print(float("-42.48"))`	`-42.48`	The string `"-42.48"` is converted into float `-42.48`
`print(float("  -24.45 \n"))`	`-24.45`	Leading and trailing spaces are trimmed and converted to float value.
`print(float("InF"))`	`inf`	`inf` represents the upper bound value of float.
`print(float("InFiNiTy"))`	`inf`	The case of text does not matter. Words used : `inf` or `infinity`
`print(float("nan"))`	`nan`	`nan` represent not a number.
`print(float("NaN"))`	`nan`	The case of text does not matter.
`print(float("CodeTantra"))`	`ValueError`	Cannot convert string "CodeTantra" to float.

3. **ord( )**: The `ord()` method returns an integer representing Unicode code point for the given Unicode character.

The syntax of `ord()` function is:

```
ord(c)
```

- `c - character string of length 1 whose Unicode code point is to be found`

```
print(ord('A'))  #  convert Unicode 'A' character to respective integer value.
print(ord('Z'))  # convert Unicode 'Z' character to respective integer value.
print(ord('a'))  # convert Unicode 'a' character to respective integer value.
print(ord('z'))  # convert Unicode 'z' character to respective integer value.

Output:
65
90
97
122
```

```
print(ord('€')) # The symbol for Euro currency is 16 bit Unicode

Output:
8364
```

If the length of the string is greater than 1, then `ord()` function results in an `TypeError`.

```
print(ord('AB')) # Will result in a TypeError

Output:
TypeError: ord() expected a character, but string of length 2 found.
```

4. **hex(x)**: The `hex()` function converts an integer to its corresponding hexadecimal string.

The syntax of `hex()` function is:

```
hex(x)

    • x - is an integer that is to be converted to hexadecimal string
Note: The returned hexadecimal string starts with prefix "0x" indicating that it is in hexadecimal form.
```

```
print(hex(45))  # Takes an int value to convert it into hexadecimal value.

Output:
'0x2d'   # respective hexadecimal value for 45. 0x prefix indicates this is a hexadecimal number
```

A non-integer results in an `TypeError: 'float' object cannot be interpreted as an integer`

```
print(hex(9.9))  # Non-integer results in a TypeError

Output:
TypeError: 'float' object cannot be interpreted as an integer
```

5. **oct(x)**: The `oct()` method takes an integer and returns its octal representation.

The syntax of `oct()` function is:

```
oct(x)

    • x - is an integer that is to be converted to an octal string
Note: The returned octal string starts with prefix "0o" indicating that it is in octal form.
```

```
print(oct(45))  # Takes an int value to convert it into octal value.

Output:
'0o55'   # respective octal value for 45. 0o prefix indicates this is an octal number
```

A non-integer results in an `TypeError: 'float' object cannot be interpreted as an integer`

```
print(ord(9.9))  # Non-integer results in a TypeError

Output:
TypeError: 'float' object cannot be interpreted as an integer
```

6. **complex(real, imag)**: The `complex()` method returns a complex number when the real and imaginary parts are provided, or it converts a string to a complex number.

The syntax of `complex() function` is:

```
complex(real, imag)
```

- real - real part. If real is omitted, it defaults to 0.
- imag - imaginary part. If imag is omitted, it default to 0.

```
print(complex(10, 3)) # Creates a complex number with real part 10 and imaginary part 3

Output:
(10 + 3j)
```

If the first parameter passed to this method is a string, it will be interpreted as a complex number.

```
print(complex("10+4j")) # Creates a complex number with real part 10 and imaginary part 3

Output:
(10 + 4j)

Note: The string passed to the complex() should be in the form real + imag
```

7)**str(x)**: The str() function is used to convert x to a string representation.

8)**eval(str)**: The eval() method parses the expression passed to this method and runs python expression (code) within the program.

9)chr(): The chr() method returns a character (a string) from an integer (that represents unicode code point of the character). This is the inverse of ord() function.

10. **tuple( )** : This function is used to convert any data type to a tuple.

```
str = "python"
print(tuple(str)) # will print output as follows
('p', 'y', 't', 'h', 'o', 'n')
```

11.**set( )** : This function is used to convert any data type to set.

```
str = "python"
print(set(str)) # will print output as follows
{'n', 't', 'p', 'h', 'y', 'o'}
```

12. **list( )** : This function is used to convert any data type to a list type.

```
str = "python"
print(list(str)) # will print output as follows
['p', 'y', 't', 'h', 'o', 'n']
```

13.**dict(d)** : This function is used to create a dictionary, but `d` must be tuple of order **(key, value)**.

```
mytuple = ((1, 'a'), (2, 'b')) # Take a tuple object to convert it into Dictionary object using dict()
function.
print(dict((y, x) for x, y in mytuple)) # will print output as follows
{'a': 1, 'b': 2}
```

Here we use **for loop** to iterate every element of tuple object and we use **dict( )** function to convert tuple elements into key-value pairs.

```
print(dict((x, y) for x, y in mytuple)) # will print output as follows
{1: 'a',  2: 'b'}
```

Question:

Write a simple program to convert given number into string, char and hexadecimal and complex number.

At the time of execution, the program should print the message on the console as:

```
Enter a value:
Enter b value:
```

For example, if the user gives the input as:

```
Enter a value: 33
Enter b value: 6
```

then the program should print the result as:

```
33
!
0x21
(33+6j)
```

Sample Input and Output:

```
Enter a value: 80
Enter b value: 20
80
P
0x50
(80+20j)
```

Solution

```
a = int(input("Enter a value: "))
b = int(input("Enter b value: "))
print(a)
print(chr(a))
print(hex(a))
print(complex(a,b))
```