INTRODUCTION TO PYTHON PROGRAMMING- MODULE 2

Dr.Thyagraju G S and Palguni GT

Contents

List

In Python, a sequence is an ordered collection of elements, where each element is identified by an index. Each element of a sequence is assigned a number – *its position or index*. The first index is zero, the second index is one, and so forth. There are three types of sequences in Python: **lists, tuples, and strings**.

2.1 List Data type

In Python, a list is a collection of ordered and mutable elements. A list can contain elements of different data types, such as integers, floats, strings, and even other lists. Lists are created by enclosing a comma-separated sequence of values within square brackets [ ].

Example:

my_list = [1, 2, 3, "four", 5.0, [6, 7, 8]]

2.1.1 Different ways to create List.

In Python, there are different ways to create a list. Here are some examples:

1. **Using square brackets:** You can create a list by enclosing a sequence of elements within square brackets. The elements can be of any data type, and can include integers, floats, strings, and even other lists.
   Example: my_list = [1, 2, 3, "four", 5.0, [6, 7, 8]]

2. **Using the list() function**: You can create a list by passing a sequence of elements to the list() function. The sequence can be any iterable object, such as a tuple or a string.
   Example:
   ```
   my_tuple = (1, 2, 3, "four", 5.0)
   my_list = list(my_tuple)
   ```

3. **Using append() method** : The append() method adds a single element to the end of the list.
   Example :

   ```
   my_list = []
   my_list.append(1)
   my_list.append(2)
   my_list.append(3)
   print(my_list) # Output: [1, 2, 3]
   ```

4. **Using extend() method** : The **extend()** method adds multiple elements to the end of the list by appending each element from the iterable that is passed as an argument.

Example :

```
my_list = []
my_list.extend([1, 2, 3])
print(my_list) # Output: [1, 2, 3]
```

In this example, the extend() method is used to add multiple elements to the end of the list at once. The elements are passed as a list argument to the extend() method.
Note that the extend() method can also add elements from other iterable objects, such as tuples or other lists.

```
my_list = []
my_list.extend((1, 2, 3))
print(my_list) # Output: [1, 2, 3]

other_list = [4, 5, 6]
my_list.extend(other_list)
print(my_list) # Output: [1, 2, 3, 4, 5, 6]
```

5. Using list comprehension: You can create a list using a list comprehension, which is a concise way to create a list based on an existing sequence.

Example: my_list = [x for x in range(10)] . This creates a list of integers from 0 to 9.

6. **Using the range() function**: You can create a list of integers using the range() function, which generates a sequence of integers based on the given arguments.

Example : my_list = list(range(1, 10, 2)). This creates a list of odd numbers from 1 to 9.

7. **Using the split() method:**

You can create a list of strings by splitting a string based on a delimiter. The split() method splits a string into a list of substrings based on a specified separator.

Example:

```
my_string = "apple,banana,orange"
my_list = my_string.split(",")
```

This creates a list of strings containing the fruits “apple”, “banana”, and “orange”.

These are some of the ways to create a list in Python. Depending on the situation, one method may be more appropriate than the others.

Creation of Empty list:

In Python empty list can be created using inbuilt function and square brackets as illustrated below :

Method1: ***using inbuilt function.***

```
empty_list = list()
print(empty_list) # Output: [ ]
```

Method 2: using square brackets.

```
empty_list = [ ]
print(empty_list) # Output: [ ]
```

Creation of List with values

To create a list with values in Python, you can define the list and assign the desired values to it. Following examples illustrates how to create a list with values:

```
my_list1 = [1, 2, 3, 4, 5]
```

In the above example, a list named my_list is created with five values: 1, 2, 3, 4, and 5. You can modify the values or add more elements to the list as needed.

You can also create a list with different types of values, such as strings, integers, or even a combination of different types:

```
my_list2 = [1, "hello", 3.14, True, "world"]
```

2.1.1.2 Lists are Mutable

In Python, a list is a collection of items that are ordered and changeable. One key feature of lists is that they are mutable, meaning that their elements can be modified after they are created.

Here's an example of how a list can be modified in Python:

```
# create a list of fruits
fruits = ['apple', 'banana', 'orange']

# add a new fruit to the list
fruits.append('strawberry')
print(fruits)  # Output: ['apple', 'banana', 'orange', 'strawberry']

# remove an item from the list
fruits.remove('banana')
print(fruits)  # Output: ['apple', 'orange', 'strawberry']

# change an item in the list
fruits[0] = 'kiwi'
print(fruits)  # Output: ['kiwi', 'orange', 'strawberry']
```

In this example, we create a list called `fruits` containing three elements: **`'apple'`, `'banana'`, and `'orange'`**. We then use the `append()` method to add a new element, `'strawberry'`, to the end of the list. Next, we use the `remove()` method to remove the `'banana'` element from the list. Finally, we use indexing to change the first element of the list from `'apple'` to `'kiwi'`.

This example demonstrates that lists are mutable because we are able to add, remove, and modify elements within the list after it has been created. This can be very useful in programming, as it allows us to dynamically update our data structures as needed.

2.1.1.3 Accessing List values (Getting individual values in lists with indexes.)

One can think List as a relationship between indices and elements. This relationship is called mapping. Each index ‘maps to’, one of the elements.

Python Supports both positive and negative indexing of list elements as illustrated below.

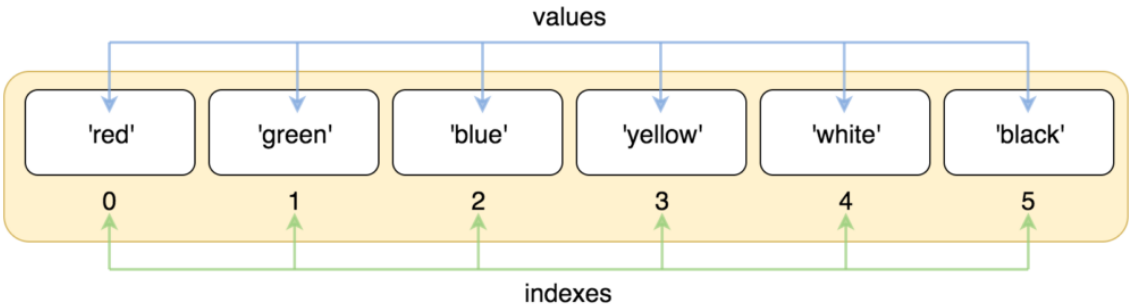


Fig: Positive Indexing [source : Link]

Positive indexing starts from 0 and goes up to n-1, where n is the length of the list. For example:

```
my_list = [10, 20, 30, 40]
print(my_list[0])    # Output: 10
print(my_list[1])    # Output: 20
print(my_list[2])    # Output: 30
print(my_list[3])    # Output: 40
```

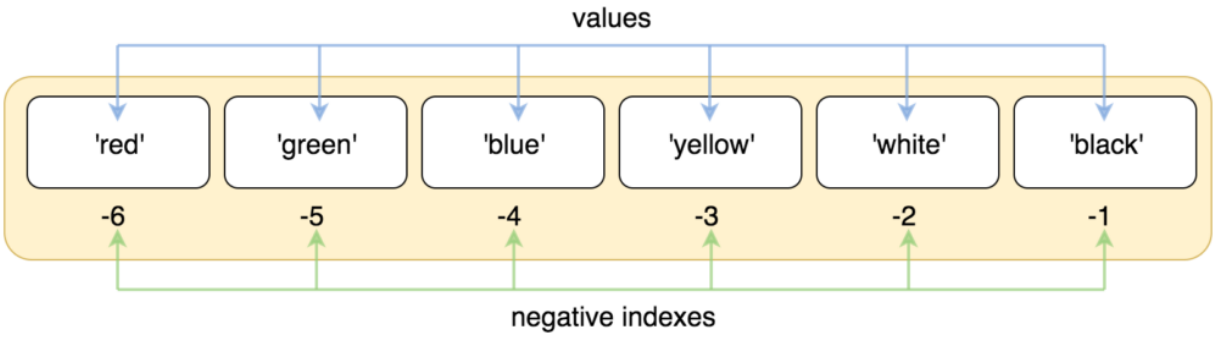


Fig: Negative Indexing [Source :Link]

Negative indexing starts from -1 and goes down to -n, where n is the length of the list. For example:

```
my_list = [10, 20, 30, 40]
print(my_list[-1])   # Output: 40
print(my_list[-2])   # Output: 30
print(my_list[-3])   # Output: 20
print(my_list[-4])   # Output: 10
```

Here are some examples that demonstrate the use of positive and negative indexing in Python lists of integers:

```
my_list = [10, 20, 30, 40, 50]

# Accessing elements using positive indexing
print(my_list[0])      # Output: 10
print(my_list[2])      # Output: 30

# Accessing elements using negative indexing
print(my_list[-1])     # Output: 50
print(my_list[-3])     # Output: 30

# Modifying elements using positive indexing
my_list[1] = 25
print(my_list)         # Output: [10, 25, 30, 40, 50]
```

```
# Modifying elements using negative indexing
my_list[-2] = 45
print(my_list)          # Output: [10, 25, 30, 45, 50]

# Slicing using positive indexing
print(my_list[1:3])     # Output: [25, 30]

# Slicing using negative indexing
print(my_list[-3:-1])   # Output: [30, 45]
```

In summary, positive indexing starts from 0 and goes up to n-1, while negative indexing starts from -1 and goes down to -n. Positive indexing is used to access or modify elements, while negative indexing is used to access elements from the end of the list. Slicing can be done using both positive and negative indexing.

2.1.1.4 Index Error

An Index Error occurs when you try to access an element in a list using an invalid index, which means the index is either negative, greater than or equal to the length of the list.
Here is an example:

```
my_list = ['apple', 'banana', 'orange']
print(my_list[3])
```

In this example, the list `my_list` has three elements, with indices 0, 1, and 2. The above code tries to access the element at index 3, which is out of range and will result in an **Index Error.**
The correct way to access the last element of this list would be to use index 2, like this:

```
print(my_list[2])
```

This will print the last element of the list, which is 'orange'.
Following example illustrates the Index error:

```
l1 =[10,20,30,40,50,60,70]
```

```
print(l1[5])
```

```
60
```

```
print(l1[7])
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1340\1955014445.py in <module>
----> 1 print(l1[7])

IndexError: list index out of range
```

2.1.1.5 List Slicing /Getting Sublists with Slices

In Python, list slicing is a way to extract a subset of elements from a list. Slicing allows you to extract a contiguous portion of a list, specified by its starting and ending indices. The syntax for slicing a list is as follows:

my_list[start:end:step]

Here, start is the index of the first element to include in the slice, end is the index of the first element to exclude from the slice, and step is the number of elements to skip between each element in the slice. Note that the end index is exclusive, which means that the element at the end index is not included in the slice.
Here are some examples of list slicing:

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Extract a slice of the first 5 elements
slice1 = my_list[0:5]
print(slice1) # Output: [1, 2, 3, 4, 5]

# Extract a slice of the elements at odd indices
slice2 = my_list[1::2]
print(slice2) # Output: [2, 4, 6, 8, 10]

# Extract a slice of the elements in reverse order
slice3 = my_list[::-1]
print(slice3) # Output: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

In the first example, we extract a slice of the first 5 elements of my_list. In the second example, we extract a slice of the elements at odd indices, starting from the second element. In the third example, we extract a slice of all the elements in reverse order.
Note that you can also use negative indices for start and end. In this case, the index is counted from the end of the list, with -1 being the index of the last element.
For example:

```
# Extract a slice of the last 3 elements
slice4 = my_list[-3:]
print(slice4) # Output: [8, 9, 10]
```

This example extracts a slice of the last 3 elements of my_list.

Example:

```
l1 = ['a','b','c','d','e','f']
print(l1[1:3])
print(l1[:4])
print(l1[3:])
print(l1[:])
```

```
['b', 'c']
['a', 'b', 'c', 'd']
['d', 'e', 'f']
['a', 'b', 'c', 'd', 'e', 'f']
```

Example:

```
x = [10,20,30,40,50]
print(x[0:2])
print(x[3:5])
print(x[:])
print(x[:-1])
print(x[:-2])
print(x[-5:])
print(x[::-1])
print(x[2::])
print(x[::])
print(x[-1:-6:-1])
```

```
[10, 20]
[40, 50]
[10, 20, 30, 40, 50]
[10, 20, 30, 40]
[10, 20, 30]
[10, 20, 30, 40, 50]
[50, 40, 30, 20, 10]
[30, 40, 50]
[10, 20, 30, 40, 50]
[50, 40, 30, 20, 10]
```

2.1.1.6 Getting lists length with len( )

In Python, the **len()** function is used to determine the length or the number of elements in a list. It returns the count of items present in the list. The len() function is a built-in function in Python that can be used with various data types, including lists, strings, tuples, and

dictionaries. It provides a convenient way to determine the size or length of a collection, allowing you to perform further operations or make decisions based on the size of the list.

Example:

```
fruits = ['apple', 'banana', 'cherry', 'durian', 'elderberry']

length = len(fruits)

print(length)
```

2.1.1.7 Changing Values in a List with Indexes

In Python, you can change the values of a list by accessing specific elements using their indexes. The index represents the position of an element in the list, starting from 0 for the first element.

```
fruits = ['apple', 'banana', 'cherry', 'durian', 'elderberry']

print(fruits)

fruits[2] = 'grape'

print(fruits)

# Output :

['apple', 'banana', 'cherry', 'durian', 'elderberry']

['apple', 'banana', 'grape', 'durian', 'elderberry']
```

In the example above, we have a list called fruits containing five elements. Initially, we print the original list. Next, we change the value at index 2 by assigning the string 'grape' to fruits[2]. Finally, we print the updated list, which reflects the change made to the value at index 2.

2.1.1.8 List Concatenation

List concatenation is the process of combining two or more lists into a single list. In Python, there are multiple ways to concatenate lists. Here are some examples:

1. **Using the '+' operator:**

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
concatenated_list = list1 + list2
print(concatenated_list)  # Output: [1, 2, 3, 4, 5, 6]
```

In the above example, we have two lists, list1 and list2. To concatenate them, we use the '+' operator, which creates a new list that contains all the elements of both list1 and list2.

2. **Using the extend() method:**

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list1.extend(list2)
print(list1)  # Output: [1, 2, 3, 4, 5, 6]
```

3. Using the append() method in a loop:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
for item in list2:
   list1.append(item)
print(list1)  # Output: [1, 2, 3, 4, 5, 6]
```

In the above example, we have two lists, list1 and list2. To concatenate them, we use a loop that iterates over each element of list2 and appends it to list1 using the append() method.

Example:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = list1+list2
print(list3)          #output : [1,2,3,4,5,6]
```

Another Example:

```
spam = [1,2,3] + ['A','B','C']
print(spam)         #output : [1,2,3, 'A','B','C']
```

2.1.1.9 List Replication

List replication is a process of creating a new list by replicating the elements of an existing list multiple times. In Python, list replication can be achieved using the '*' operator. Here are some examples:

1. Replicating a list multiple times:

```
original_list = [1, 2, 3]
replicated_list = original_list * 3
print(replicated_list) # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In the above example, we have an original list original_list with three elements. We replicate this list three times using the '*' operator, which creates a new list replicated_list that contains the elements of original_list repeated three times.

2. Creating a list of a specific length with a single element repeated:

```
single_element_list = [0] * 5
print(single_element_list) # Output: [0, 0, 0, 0, 0]
```

In the above example, we create a new list single_element_list that contains the element 0 repeated five times using the '*' operator. This is a useful technique when you need to create a list of a specific length with a single element repeated.

3. Replicating a list using a variable:

```
original_list = [1, 2, 3]
n = 4
replicated_list = original_list * n
print(replicated_list) # Output: [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

In the above example, we use a variable n to specify the number of times we want to replicate the original_list. The replicated_list is created by multiplying original_list by n using the '*' operator.

2.1.1.10 Removing values from lists with del statement

In Python, you can remove elements from a list using the del statement. The del statement is a powerful Python keyword that allows you to delete an object, such as a variable or an element in a list. Here are some examples of removing values from lists using the del statement:

1. **Removing an element by index:**

```
my_list = [1, 2, 3, 4, 5]
del my_list[1:4]
print(my_list) # Output: [1, 5]
```

In the above example, we have a list my_list with five elements. We use the del statement to remove the elements at indices 1, 2, and 3, which have the values 2, 3, and 4. We do this by using slicing notation [1:4], which selects the elements at indices 1, 2, and 3. After deleting the elements, the list contains the remaining elements [1, 5].

2. Removing multiple elements by slicing:

```
my_list = [1, 2, 3, 4, 5]
del my_list[1:4]
print(my_list) # Output: [1, 5]
```

In the above example, we have a list my_list with five elements. We use the del statement to remove the elements at indices 1, 2, and 3, which have the values 2, 3, and 4. We do this by using slicing notation [1:4], which selects the elements at indices 1, 2, and 3. After deleting the elements, the list contains the remaining elements [1, 5].

3. **Removing all elements:**

```
my_list = [1, 2, 3, 4, 5]
del my_list[:]
print(my_list) # Output: []
```

In the above example, we have a list my_list with five elements. We use the del statement to remove all elements of the list by slicing the entire list with [:]. After deleting all elements, the list is empty, [].

Note that the del statement permanently deletes the elements from the list, so you should be careful when using it. Make sure you have a copy of the original list or are sure that you no longer need the deleted elements.
The del statement can also be used on a simple variable to delete it, as if it were an “unassignment” statement. If you try to use the variable after deleting it, you will get a NameError error because the variable no longer exists.

In practice, you almost never need to delete simple variables. The del statement is mostly used to delete values from lists.

Example:

```
ham = ['a','ant',2015,'rat','apple']
print(ham)
```

```
['a', 'ant', 2015, 'rat', 'apple']
```

```
del ham[2]  # deleting single element
print(ham)
```

```
['a', 'ant', 'rat', 'apple']
```

2nd element gets deleted

```
del ham[0:2] # deleting elements in the range
print(ham)
```

```
['rat', 'apple']
```

Elements from index number 0 to 1 gets deleted.

```
del ham # deleting entire list
print(ham)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1340\3203663453.py in <module>
      1 del ham # deleting entire list
----> 2 print(ham)

NameError: name 'ham' is not defined
```

If you try to print **ham** after deleting it completely, you will get a NameError error because the list no longer exists.

2.1.2 Working with Lists

Using for loops with Lists:

Using for loops with lists is a common technique in programming to iterate over the elements of a list and perform certain operations on each element. It allows you to easily process each item in a list without manually accessing them one by one.
The basic syntax for a for loop with a list is as follows:

```
for item in my_list:
    # Code block to execute for each item
    # ...
# Code outside the loop
# ...
```

Here's a simple example to illustrate how to use a for loop with a list:

```
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(fruit)
print("Loop finished.")
```

The output of the above code would be:

```
apple
banana
orange
Loop finished.
```

The in and not in operator

The in and not in operators in Python are used to check whether a certain element exists or does not exist in a list. These operators return a Boolean value (True or False) based on the evaluation of the condition.
Here's the syntax for using the in and not in operators with lists:

```
element in my_list  # Returns True if 'element' is found in 'my_list'
element not in my_list  # Returns True if 'element' is not found in 'my_list'
```

Let's look at some examples to illustrate their usage:

```
fruits = ["apple", "banana", "orange"]
print("apple" in fruits)  # True
print("grape" in fruits)  # False
print("banana" not in fruits)  # False
print("kiwi" not in fruits)  # True
```

The Multiple Assignment Trick

The "multiple assignment trick" in Python allows you to assign multiple values from a list to multiple variables in a single line of code. It is particularly useful when working with lists of known length or when you want to access individual elements of a list directly.
To use the multiple assignment trick, you need to have the same number of variables on the left side of the assignment operator (=) as the number of elements in the list on the right side. The variables will be assigned values in the order they appear in the list.
Here's an example to illustrate the multiple assignment trick:

```
fruits = ["apple", "banana", "orange"]
fruit1, fruit2, fruit3 = fruits
print(fruit1)  # "apple"
print(fruit2)  # "banana"
print(fruit3)  # "orange"
```

It's important to note that the number of variables on the left side must match the number of elements in the list; otherwise, a ValueError will occur. Also, if the list contains more elements than the number of variables, the excess elements will not be assigned to any variables. Conversely, if there are fewer elements in the list, a ValueError will be raised.

```
# Example with more elements in the list than variables
fruits = ["apple", "banana", "orange", "kiwi"]

fruit1, fruit2 = fruits  # ValueError: too many values to unpack

# Example with fewer elements in the list than variables
fruits = ["apple", "banana"]

fruit1, fruit2, fruit3 = fruits  # ValueError: not enough values to unpack
```

2.1.3 Augmented Assignment Operators

When assigning a value to a variable, you will frequently use the variable itself. For example, after assigning 42 to the variable spam, you would increase the value in spam by 1 with the following code:

```
1  x = 72
2  x = x+1
3  x
```

```
73
```

As a shortcut, you can use the augmented assignment operator += to do the same thing:

```
1  x = 72
2  x+=1
3  x
```

```
73
```

There are augmented assignment operators for the +, -, *, /, and % operators, described in table below:

Augmented assignment statement	Equivalent Assignment
s +=1	s = s+1
s -=1	s = s-1
s *=1	s = s*1
s /=1	s = s/1
s %=1	s =s%1

The += operator can also do string and list concatenation, and the *= operator can do string and list replication. Enter the following into the interactive shell:

s = ‘Hello’
s += ‘Python’
print(s) # ‘HelloPython’

```
p =[ ‘Sudhanvitha’]
p *= 3
print(p)  # [ ‘Sudhanvitha’, ‘Sudhanvitha’, ‘Sudhanvitha’]
```

2.1.4 List Methods

A method is the same thing as a function, except it is “called on” a value. For example, if a list value were stored in name, you would call the **index()** list method on that list like so: **name.index('Pal')**. The method part comes after the value, separated by a period. Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list. Some of the most useful list methods are described below:

append()

append() is a built-in method in Python’s list class that adds a new element to the end of the list. The syntax for using append() is straightforward:

```
list_name.append(element)
```

Here, list_name refers to the name of the list you want to modify, and element is the value you want to append to the list.

Here’s an example that demonstrates the use of append():

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

extend( )

The extend() method is used to append multiple elements from an iterable (such as a list, tuple, or string) to the end of a list.

```
fruits = ['apple', 'banana', 'cherry']
additional_fruits = ['orange', 'mango']
```

```
fruits.extend(additional_fruits)
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'orange', 'mango']
```

insert()

The insert() method is used to insert an element at a specific index in a list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.insert(1, 'orange')
print(fruits)  # Output: ['apple', 'orange', 'banana', 'cherry']
```

remove()

The remove() method is used to remove the first occurrence of a specified element from a list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.remove('banana')
print(fruits)  # Output: ['apple', 'cherry']
```

pop()

The pop() method is used to remove and return an element from a specific index in a list. If no index is specified, it removes and returns the last element.

```
fruits = ['apple', 'banana', 'cherry']
removed_fruit = fruits.pop(1)
print(removed_fruit)  # Output: 'banana'
print(fruits)       # Output: ['apple', 'cherry']
```

index()

The index() method is used to find the index of the first occurrence of a specified element in a list.

```
fruits = ['apple', 'banana', 'cherry']
index = fruits.index('banana')
print(index)  # Output: 1
```

count()

The count() method is used to count the number of occurrences of a specified element in a list.

```
fruits = ['apple', 'banana', 'cherry', 'banana']
count = fruits.count('banana')
print(count)  # Output: 2
```

sort()

The sort() method is used to sort the elements of a list in ascending order. It modifies the original list.

```
numbers = [3, 1, 4, 1, 5, 9, 2]
numbers.sort()
print(numbers)  # Output: [1, 1, 2, 3, 4, 5, 9]
```

reverse()

The reverse() method is used to reverse the order of elements in a list. It modifies the original list.

```
fruits = ['apple', 'banana', 'cherry']
fruits.reverse()
print(fruits)  # Output: ['cherry', 'banana', 'apple']
```

copy()

The copy() method is used to create a shallow copy of a list, which means it creates a new list with the same elements. Modifying the original or copied list does not affect the other.

```
fruits = ['apple', 'banana', 'cherry']
copied_fruits = fruits.copy()
fruits.append('orange')
print(fruits)        # Output: ['apple', 'banana', 'cherry', 'orange']
print(copied_fruits) # Output: ['apple', 'banana', 'cherry']
```

clear()

The clear() method is used to remove all the elements from a list, making it empty.

```
fruits = ['apple', 'banana', 'cherry']
fruits.clear()
print(fruits)  # Output: []
```

2.1.5 Example Program: Magic 8 Ball with a List .

The Magic 8 Ball problem refers to a hypothetical situation in computer science where a program needs to make a decision based on incomplete or ambiguous information. It is named after the Magic 8 Ball toy, which is a popular fortune-telling device that provides random answers to yes-or-no questions. Following program illustrates the Magic 8 Ball for yes or no questions.

```
import random
messages = ['It is certain',
 'It is decidedly so',
 'Yes definitely',
 'Reply hazy try again',
 'Ask again later',
 'Concentrate and ask again',
 'My reply is no',
 'Outlook not so good',
 'Very doubtful']

print(messages[random.randint(0, len(messages) - 1)])
```

2.2 Strings

In Python, a string is a sequence of characters enclosed within either single quotes (‘ ‘) or double quotes (” “). Strings are one of the basic data types in Python, and they are used to store and manipulate text-based data.

Example : ‘Hello World!’ , “Python Programming“ , ”’apple”’

Creating String:

In Python, you can create a string by enclosing a sequence of characters within either single quotes (‘ ‘) or double quotes (” “).

Here are some examples:

```
my_string_1 = 'This is a string created with single quotes'
my_string_2 = "This is a string created with double quotes"
```

In these examples, my_string_1 and my_string_2 are both strings that contain the text “This is a string created with single quotes” and “This is a string created with double quotes”, respectively.

You can also create strings that contain special characters such as newlines (\n) and tabs (\t):

```
my_string_3 = "This string\n has a newline"
my_string_4 = "This string\t has a tab"
```

In these examples, my_string_3 contains a newline character, which causes the text “has a newline” to appear on a new line, and my_string_4 contains a tab character, which causes a tab space before the text “has a tab”.

Example :

```
var1 = 'Hello World!'
var2 = "Python Programming"
var3 = '''Water Melon'''
print(var1)
print(var2)
print(var3)
```

```
Hello World!
Python Programming
Water Melon
```

Triple quotes can extend multiple lines as illustrated below:

```
my_string = """
                Hello, Welcome to
                the World of Python
                Programming '
            """
print(my_string)
```

```
                Hello, Welcome to
                the World of Python
                Programming '
```

Length of a string

In Python, you can use the built-in len() function to get the length of a string, which returns the number of characters in the string. Here's an example:

```
my_string = "Hello, World!"
print(len(my_string)) # Output: 13
```

In this example, the **len()** function is called on the **my_string** string, which returns the number of characters in the string (which is 13 in this case).
Note that the **len()** function counts all characters in the string, including spaces and special characters. For example:

```
my_string = "This string\n has a newline"
print(len(my_string)) # Output: 22
```

In this example, the my_string string contains a newline character (\n), which is counted as a single character by the len() function.
You can use the length of a string in various ways in your code, such as checking if a string is empty, looping through the characters in a string, or performing string slicing operations.
You can find the length of a string in Python without using the built-in len() function by iterating through the characters in the string using a loop and counting them. Here's an example:

```
my_string = "Hello, World!"
count = 0
for char in my_string:
    count += 1
print(count) # Output: 13
```

In this example, a variable count is initialized to 0, and then a for loop is used to iterate through each character in the my_string string. For each character, the count variable is incremented by 1.

After the loop has finished, the count variable contains the length of the string (which is 13 in this case).
Note that this method is not as efficient as using the built-in len() function, especially for longer strings, because it involves iterating through each character in the string. However, it can be a useful exercise in understanding how strings are represented and manipulated in Python.
One can get last letter of string using **len( )** function as illustrated below :

```
length = len(fruits)
last = fruits[length -1]
print(last)
```

```
n
```

To get the last letter of a string in Python, you can use string indexing with a negative index of -1, which refers to the last character in the string. Here's an example:

```
my_string = "Hello, World!"

last_letter = my_string[-1]

print(last_letter) # Output: !
```

In this example, the **my_string** string is indexed using a negative index of -1, which refers to the last character in the string (which is the exclamation mark in this case). The **last_letter** variable is then assigned this character using string indexing.
Note that this method assumes that the string is not empty. If the string is empty, attempting to index it with **-1** will result in an **IndexError**. To avoid this, you should first check if the string is non-empty before indexing it.

String Indexing

In Python, you can use indexing to access individual characters in a string. String indexing is done by specifying the position of the character within square brackets **([])** immediately following the string. There are two types of indexing in Python: positive indexing and negative indexing.

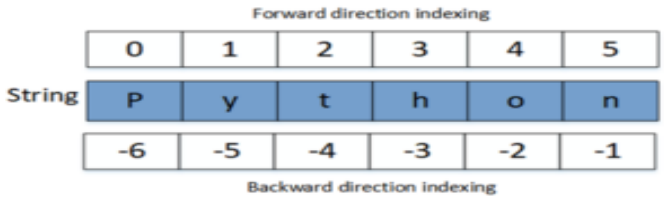
```
my_string = "Hello, World!"
print(my_string[0]) # Output: H
print(my_string[1]) # Output: e
print(my_string[6]) # Output: ,
```

In this example, the my_string string is indexed using positive indices. The first print statement outputs the first character of the string (which is "H"), the second print statement outputs the second character of the string (which is "e"), and the third print statement outputs the seventh character of the string (which is ",").

Negative indexing, on the other hand, starts from the rightmost character of the string, with the index of the last character being -1, the second-to-last character being -2, and so on. For example:

```
my_string = "Hello, World!"
print(my_string[-1]) # Output: !
print(my_string[-2]) # Output: d
print(my_string[-7]) # Output:
```

In this example, the my_string string is indexed using negative indices. The first print statement outputs the last character of the string (which is "!"), the second print statement outputs the second-to-last character of the string (which is "d"), and the third print statement outputs the seventh-to-last character of the string (which is a space).
In Python string can be indexed in forward(start to end) and backward(end to start) direction as illustrated in the figure below :



Consider a string **name = " Sophia"**. Here string Sophia is made up of 6 characters. Each character of the string can be accessed by using either forward indexing or backward indexing as illustrated below:

String Traversal

String traversal in Python involves iterating over each character of a string. This can be done in both forward and reverse order using loops. Forward traversal using a while loop:

```
my_string = "hello world"
i = 0
while i < len(my_string):
    print(my_string[i])
    i += 1
# Output:
h
e
l
l
o

w
o
r
l
d
```

In this example, we initialize the index i to 0 and use a while loop to iterate through each character of the string. The loop continues as long as i is less than the length of the string. At each iteration, we print the character at the current index i and then increment i by 1 to move to the next character.

Forward traversal using a for loop:

```
my_string = "hello world"
for char in my_string:
    print(char)
# output
h
e
l
l
o

w
o
r
l
d
```

In this example, we use a for loop to iterate through each character of the string. The loop variable char takes on the value of each character in the string in turn, allowing us to access and print each character. Reverse traversal using a while loop:

```
my_string = "hello world"
i = len(my_string) - 1
while i >= 0:
    print(my_string[i])
    i -= 1
#Output:
d
l
r
o
w

o
l
l
e
h
```

In this example, we initialize the index i to the last index of the string (which is the length of the string minus one) and use a while loop to iterate through each character of the string in reverse order. The loop continues as long as i is greater than or equal to 0. At each iteration, we print the character at the current index i and then decrement i by 1 to move to the previous character.

Reverse traversal using a for loop:

```
my_string = "hello world"
for char in reversed(my_string):
    print(char)
#output
d
l
r
o
w

o
l
l
e
h
```

In this example, we use the built-in reversed() function to reverse the order of the characters in the string, and then use a for loop to iterate through each character in the reversed order. The loop variable char takes on the value of each character in the string in turn, allowing us to access and print each character.
Lists aren't the only data types that represent ordered sequences of values.For example, strings and lists are actually similar, if you consider a string to be a "list" of single text characters. Many of the things you can do with lists can also be done with strings: ***indexing; slicing; and using them with for loops, with len(), and with the in and not in operators***.

String Slicing

String slicing is a technique in Python that allows you to extract a portion of a string by specifying its start and end positions. This is done using the slice operator :. The basic syntax for string slicing is as follows:

string[start:end:step]

start is the index of the first character to include in the slice (inclusive)
end is the index of the last character to include in the slice (exclusive)
step is the number of characters to skip between each included character (optional)

Here are some examples of string slicing:

```
my_string = "Hello, World!"
# Extract the first 5 characters
print(my_string[0:5])    # Output: "Hello"
```

In the above example, the slice operator 0:5 is used to extract the first 5 characters of the string "Hello, World!". This includes the characters at indices 0 through 4 (inclusive).

Extract the characters from index 7 to the end
print(my_string[7:]) # Output: "World!"
In the above example, the slice operator 7: is used to extract the characters from index 7 (the 'W' in "World!") to the end of the string. This includes all characters starting at index 7 and continuing to the end of the string.

Extract the last 6 characters
print(my_string[-6:]) # Output: "World!"
In the above example, negative indexing and the slice operator -6: is used to extract the last 6 characters of the string "Hello, World!". This includes all characters starting 6 positions from the end of the string and continuing to the end of the string.

Extract every other character
print(my_string[::2]) # Output: "Hlo ol!"

In the above example, the slice operator ::2 is used to extract every other character of the string "Hello, World!". This includes the first character, then skips the next character, then includes the third character, and so on.

Reverse the string
print(my_string[::-1]) # Output: "!dlroW ,olleH

In the above example, negative step and the slice operator [::-1] is used to reverse the string "Hello, World!". This includes all characters in the string, but they are extracted in reverse order because the step is -1.

Example :

```
strx = "Context Aware Computing"
print(len(strx))
print(strx[:])
print(strx[0:22])
print(strx[0:10])
print(strx[5:12])
print(strx[:10])
print(strx[10:23])
print(strx[-23:-1])
print(strx[::-1])
```

```
23
Context Aware Computing
Context Aware Computin
Context Aw
xt Awar
Context Aw
are Computing
Context Aware Computin
gnitupmoC erawA txetnoC
```

Strings are immutable

In Python, strings are immutable objects, which means that once a string is created, it cannot be modified. Immutable objects are objects whose state cannot be changed once they are created.

When we modify a string in Python, what actually happens is that a new string is created with the desired modifications, while the original string remains unchanged. This is different from mutable objects like lists, where you can modify their elements in place.

Here is an example to demonstrate string immutability:

```
my_string = "hello"
my_string[0] = "H"    # This will cause an error
```

In this example, we try to modify the first character of the string "hello" to be uppercase. However, this will result in a TypeError, because strings are immutable and cannot be modified in place.

Instead, we need to create a new string with the desired modifications. Here is an example:

```
my_string = "hello"
new_string = "H" + my_string[1:]
print(new_string)    # Output: "Hello"
```

In this example, we create a new string by concatenating the uppercase letter "H" with a slice of the original string that starts at index 1 and goes to the end. This creates a new string "Hello" that has the desired modification.

Example : The proper way to **'mutate'** a string is to **slice** and **concatenate** to build a new string by copying from parts of the old string as illustrated below :

```
name = "Sophia a Robot"
newname1 = name[0:7] + 'the' + name[8:14]
newname2 = "Palu" + " is not a" + name[8:14]
print(newname1)
print(newname2)
```

```
Sophia the Robot
Palu is not a Robot
```

```
# Substituting using concatenation
p = 'Hello, Python World!'
np = 'J'+ p[1:]
print(np)
```

```
Jello, Python World!
```

Strings Looping and Counting

Looping over a string: To loop over a string in Python, you can use a for loop. In each iteration of the loop, the loop variable will take on the value of the next character in the string. Here is an example:

```
my_string = "hello"
for char in my_string:
    print(char)
# Output:
h
e
l
l
o
```

Counting occurrences of a character in a string: To count the number of occurrences of a specific character in a string, you can use the count() method of the string. This method takes a single argument, which is the character to count. Here is an example:

```
my_string = "hello"
count = my_string.count('l')
print(count)    # Output: 2
```

Counting occurrences of a substring in a string: To count the number of occurrences of a substring in a string, you can use a loop and the count() method. Here is an example:

```
my_string = "hello world"
substring = "l"
count = 0
for char in my_string:
    if char == substring:
        count += 1
print(count)    # Output: 3
```

In this example, we loop over each character in the string and increment a counter if the character matches the substring we are searching for.

Example :

```
word = 'Strawberry'
count = 0
for letter in word:
    if letter == 'r':
        count = count + 1
print(count)
```

```
3
```

Example : usage of in operator for strings

The in operator is a built-in operator in Python that is used to check whether a given value is present in a sequence (such as a string, list, or tuple). When used with a string, the in operator checks whether a substring is present in the string. Here are some examples:

```
# Check if a substring is present in a string
my_string = "hello world"
substring = "world"
if substring in my_string:
    print("Substring found")
else:
    print("Substring not found")

# Check if a character is present in a string
my_string = "hello"
char = "l"
if char in my_string:
    print("Character found")
else:
    print("Character not found")

#Output
Substring found
Character found
```

In the first example, we use the in operator to check whether the substring “world” is present in the string “hello world”. Since it is present, the output is “Substring found”.
In the second example, we use the in operator to check whether the character “l” is present in the string “hello”. Since it is present twice, the output is “Character found”.
Note that the in operator is case-sensitive. For example, if you search for the substring “World” in the string “hello world”, it will not be found because the capitalization does not match.

Example :

```
'straw' in 'strawberry'
```

```
True
```

```
'apple' in 'strawberry'
```

```
False
```

String Comparison using ==, < and > operators

In Python, you can compare strings using the ==, <, and > operators, just like you can with other data types such as numbers.
Here's how each operator works for string comparison:

= = (equal to) operator:

The == operator checks whether two strings have the same content. It returns True if the strings are equal and False otherwise. Here's an example:

```
string1 = "hello"
string2 = "hello"
if string1 == string2:
   print("The strings are equal")
else:
   print("The strings are not equal")
#Output: The strings are equal
```

< (less than) and > (greater than) operators:

The < and > operators compare two strings lexicographically (i.e., in dictionary order). They return True if the first string is less than or greater than the second string, respectively, and False otherwise. Here's an example:

```
string1 = "apple"
string2 = "banana"
if string1 < string2:
   print("string1 comes before string2 in dictionary order")
else:
   print("string2 comes before string1 in dictionary order")

#Output: string1 comes before string2 in dictionary order
```

Note that the comparison of strings is case-sensitive, meaning that uppercase letters come before lowercase letters in the ASCII table. Also, the <= and >= operators can also be used for string comparison, which check whether the first string is less than or equal to, or greater than or equal to the second string, respectively.

Useful String methods

Strings are an example of Python objects. An object contains both data (the actual string itself) and methods which are effectively functions that are built into the object and are available to any instance of the object. Using the command **dir(str)** one can list all the methods supported by python as illustrated below:

```
print(dir(str))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__i
nit_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new_
_', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'ex
pandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'is
digit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'joi
n', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 's
trip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Table : String Methods and Description

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
find()	Searches the string for a specified value and returns the position of where it was found
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title

isupper()	Returns True if all characters in the string are upper case
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
replace()	Returns a string where a specified value is replaced with a specified value
rjust()	Returns a right justified version of the string
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case

[Source : https://www.w3schools.com/python/python_ref_string.asp]

Some of the most useful string methods are described below with coding examples

1. capitalize ():

The capitalize() method is a built-in method in Python that converts the first character of a string to uppercase and leaves the rest of the string unchanged. If the first character of the string is already in uppercase, then the method has no effect on that character.
Here's an example:

```
my_string = "hello world"
capitalized = my_string.capitalize()
print(capitalized)   # Output: "Hello world"
my_string = "hello world"
capitalized = my_string.capitalize()
print(capitalized)   # Output: "Hello world"
```

In this example, the capitalize() method is used to convert the first character of the string "hello world" to uppercase, resulting in the string "Hello world".
Note that capitalize() does not modify the original string, but returns a new string with the first character capitalized. Also, if the original string is empty, the capitalize() method will return an empty string.

2. lower()

The lower() method is a built-in string method in Python that returns a new string with all the alphabetic characters in the original string converted to lowercase.
Here's an example:

```
my_string = "Hello, World!"
lowercase_string = my_string.lower()
print(lowercase_string)
# Output: hello, world!
```

In this example, the lower() method is called on the my_string variable and the returned value is assigned to the lowercase_string variable. The original string my_string is not modified, instead a new string is created with all alphabetic characters in lowercase.
Note that lower() method only works with alphabetic characters, and any non-alphabetic characters (such as punctuation, numbers, and spaces) are not affected by the method.

Example :

```
# Lower
word = input("Enter a word in upper case: ")
new_word = word.lower()
print("Entered Word : ", word)
print("New word : ",new_word)
```

```
Enter a word in upper case: BENGALURU
Entered Word :  BENGALURU
New word :  bengaluru
```

3. upper()

The upper() method is a built-in string method in Python that is used to convert all the lowercase characters in a string to uppercase characters. The method takes no arguments and returns a new string with all the characters in uppercase.
Here's an example:

```
text = "hello, world!"
uppercase_text = text.upper()
print(uppercase_text)
# Output : HELLO, WORLD!
```

In the above example, the upper() method is used to convert all the lowercase characters in the string text to uppercase characters. The resulting string is then stored in the variable uppercase_text and printed to the console.
It's important to note that the upper() method does not modify the original string, but instead returns a new string with all the characters in uppercase. If you want to modify the original string, you can assign the result of the upper() method back to the original variable:

```
text = "hello, world!"
text = text.upper()
print(text)
# Output : HELLO, WORLD!
```

Example :

```
# upper ()
word = input("Enter a word in lower case")
new_word = word.upper()
print("Entered Word : ", word)
print("New word : ",new_word)
```

```
Enter a word in lower caseabcd efgh ijklmn
Entered Word :  abcd efgh ijklmn
New word :  ABCD EFGH IJKLMN
```

4. casefold():

Actually, the casefold() method in Python is used to convert a string into a lowercase string, just like the lower() method. However, the casefold() method is more aggressive in its conversion and is often preferred over the lower() method when dealing with case-insensitive string comparisons. The casefold() method removes all case distinctions from a string and converts it into a form that is suitable for case-insensitive comparisons. This method also handles some special cases, such as the German "sharp s" character (ß), which is converted to "ss" when casefolded.
Here's an example:

```
text = "Hello, World!"
lowercase_text = text.casefold()
print(lowercase_text)
#output : hello, world!
```

In the above example, the casefold() method is used to convert the string text to lowercase. The resulting string is then stored in the variable lowercase_text and printed to the console.
Just like the lower() method, the casefold() method does not modify the original string, but instead returns a new string with all the characters in lowercase. If you want to modify the original string, you can assign the result of the casefold() method back to the original variable:

```
text = "Hello, World!"
text = text.casefold()
print(text)
# Output: hello, world!
```

Example:

```
String_word = input("Enter any word : ")
print(String_word.casefold())
```

```
Enter any word : BENGALURU
bengaluru
```

4. center() :

The center() method is a built-in string method in Python that returns a centered string. It takes two arguments: **width,** which specifies the total width of the resulting string, and an optional **fillchar** argument that specifies the character to be used for padding.

Here's the syntax for center() method:

```
string.center(width, fillchar)
```

The width argument is required, and must be a positive integer. The fillchar argument is optional, and must be a string of length 1. If fillchar is not specified, a space character is used for padding.
Here's an example of how to use the center() method:

```
string = "Hello"
width = 10
fillchar = "*"
centered_string = string.center(width, fillchar)
print(centered_string)
# Output: **Hello***
```

In this example, the original string "Hello" is centered in a string of width 10 with the fill character *, resulting in the string **Hello***.

```
String_word = input("Enter any word : ")
print(String_word.center(20))
```

```
Enter any word : google
       google
```

5. swapcase()

swapcase() is a built-in string method in Python that returns a new string where the case of each character in the original string is swapped. In other words, all uppercase characters are converted to lowercase, and all lowercase characters are converted to uppercase.
Here's the syntax for swapcase() method:

string.swapcase()

Here's an example of how to use the swapcase() method:

string = "Hello World"
swapped_string = string.swapcase()
print(swapped_string)
#Output: hELLO wORLD

In this example, the original string "Hello World" is swapped using the swapcase() method, resulting in the string "hELLO wORLD".

```
# Swap Case
stmt = 'Welcome to PYTHON programming'
print("Given Statement : ", stmt)
print("Statement with case swaped: ", stmt.swapcase())
```

```
Given Statement :  Welcome to PYTHON programming
Statement with case swaped:  wELCOME TO python PROGRAMMING
```

6. find()

find() is a built-in string method in Python that returns the lowest index of the first occurrence of a specified substring within a string. If the substring is not found, it returns -1. Here's the syntax for find() method:

string.find(substring, start, end)

The substring argument is required, and specifies the substring to be searched within the string. The start and end arguments are optional, and specify the starting and ending index positions for

the search. If start is not specified, the search starts from the beginning of the string. If end is not specified, the search goes until the end of the string.

Here's an example of how to use the find() method:

```
string = "Hello World"
substring = "World"
index = string.find(substring)
print(index)
# output : 6
```

In this example, the find() method is used to search for the substring "World" within the string "Hello World". The method returns the lowest index of the substring, which is 6. If the substring is not found within the string, the find() method returns -1:

```
string = "Hello World"
substring = "Universe"
index = string.find(substring)
print(index)
# Output: -1
```

Example:

```
# find()
word = input("Enter a word ")
ch = input(" Enter a character to find:  ")
index = word.find(ch)
print("The index of the first occurence of character {0} is {1}".format(ch,index))
```

```
Enter a word strawberyy
 Enter a character to find:  a
The index of the first occurence of character a is 3
```

Example :

```
# find () to find substrting
word = input("Enter a word ")
sb = input(" Enter a substring to find:  ")
index = word.find(sb)
print("The index of the first occurence of substring {0} is {1}".format(sb,index))
```

```
Enter a word bananas
 Enter a substring to find:  an
The index of the first occurence of substring an is 1
```

7. count()

count() is a built-in string method in Python that returns the number of occurrences of a specified substring within a string. Here's the syntax for count() method:

string.count(substring, start, end)

The substring argument is required, and specifies the substring to be counted within the string. The start and end arguments are optional, and specify the starting and ending index positions for the count. If start is not specified, the count starts from the beginning of the string. If end is not specified, the count goes until the end of the string. Here's an example of how to use the count() method:

```
string = "Hello World"
substring = "l"
count = string.count(substring)
print(count)
# output : 3
```

In this example, the count() method is used to count the number of occurrences of the substring "l" within the string "Hello World". The method returns the number of occurrences, which is 3.

You can also use the count() method to count the occurrences of a substring within a specific range of the string:

```
string = "Hello World"
substring = "l"
start = 3
end = 8
count = string.count(substring, start, end)
print(count)
#Output : 1
```

In this example, the count() method is used to count the number of occurrences of the substring "l" within the range of the string "lo Wo", which starts from index 3 and ends at index 8. The method returns the number of occurrences within the range, which is 1.

Example :

```
# Count
var1 = 'Welcome to Python Programming'
var2 = "TextCon Warriors"
var3 = "Contextual Warriors"
print(var1.count('W'))
print(var2.count('r'))
print(var3.count('a',0,10)) # Count 'a' from 0 index to 9 index  in var3
```

```
1
3
1
```

8. strip(), lstrip() and rstrip()

strip(), lstrip(), and rstrip() are built-in string methods in Python that remove leading and/or trailing characters from a string. strip() removes leading and trailing whitespace characters (spaces, tabs, newlines) from a string. Here's the syntax for strip() strip(), lstrip() , and rstrip() methods:

```
string.strip([characters])
string.lstrip([characters])
string.rstrip([characters])
```

The characters argument is optional, and specifies the characters to be removed from both ends of the string. If characters is not specified, it removes whitespace characters by default.

```
# strip(),lstrip(),rstrip() method
line = '     YES I CAN DO IT       '
print(line.strip())
print(line.lstrip())
print(line.rstrip())
```

```
YES I CAN DO IT
YES I CAN DO IT
     YES I CAN DO IT
```

9. startswith() and endswith()

startswith() and endswith() are built-in string methods in Python that are used to check whether a string starts with or ends with a specific substring, respectively.

- startswith() returns True if the string starts with the specified substring, and False otherwise.
- endswith() returns True if the string ends with the specified substring, and False otherwise.

Here's the syntax for startswith() and endswith() method:

```
string.startswith(substring, start, end)
string.endswith(substring, start, end)
```

Example:

```
# startswith and ends with
line = 'YES I CAN DO IT'
print(line.startswith('YES'))
print(line.endswith('IT'))
```

```
True
True
```

10. split()

split() is a built-in string method in Python that is used to split a string into a list of substrings based on a specified separator.
Here's the syntax for **split()** method:

```
string.split(sep=None, maxsplit=-1)
```

The sep argument is optional, and specifies the separator to use for splitting the string. If sep is not specified, the default separator is whitespace.

The maxsplit argument is optional, and specifies the maximum number of splits to be performed.

If maxsplit is not specified or set to -1, all possible splits are performed.

Here's an example of how to use the split() method:

```
string = "Hello World"
words = string.split()
print(words)

#output : ['Hello', 'World']
```

Example:

```
string = "apple,banana,orange"
fruits = string.split(",")
print(fruits)
#output: ['apple', 'banana', 'orange']
```

Example:

```
# split()
multilines = ''' Dear Students,
                Good Morning,
                Today is Python Day'''
print(multilines.split())     # splitting with space as delimieter
print(multilines.split('\n')) # splitting with '\n' as delimieter
print(multilines.split(','))  # splitting with ',' as delimieter
```

```
['Dear', 'Students,', 'Good', 'Morning,', 'Today', 'is', 'Python', 'Day']
[' Dear Students, ', '                Good Morning,', '                Today is Python Day']
[' Dear Students', ' \n                Good Morning', '\n                Today is Python Day']
```

Example:

```
# Reading multiple values using split()
n1,n2,n3 = input("Enter 3 integer values : ").split()
n1 = int(n1)
n2 = int(n2)
n3 = int(n3)
print("Entered 3 integer values are :",n1,n2,n3)
```

```
Enter 3 integer values : 20 30 40
Entered 3 integer values are : 20 30 40
```

11. join()

`join()` is a built-in method in Python that can be used to concatenate a sequence of strings into a single string. The method takes an iterable, such as a list or tuple, and concatenates each element in the iterable into a single string with a separator between them. Here's an example of how to use `join()`

```
words = ["hello", "world", "!"]
separator = " "

sentence = separator.join(words)
print(sentence)
#output: hello world !
```

In this example, we create a list of words and a separator string. We then use the join() method to concatenate the words in the list into a single string, with the separator between each word. The resulting string is assigned to the sentence variable and printed to the console. Note that the join() method can be used with any iterable that contains strings, not just lists. Additionally, the separator string can be any string value, including an empty string.

```
# Join method
dolls = ['All','dancing','dolls','are','not','Robots']
delimeter =' '
delimeter.join(dolls)
```

```
'All dancing dolls are not Robots'
```

Python String Operators

Python provides several operators that can be used with strings. Here are some of the most commonly used string operators in Python:

1. **+ (concatenation operator):** This operator is used to concatenate two or more strings. For example:

```
string1 = "Hello"
string2 = "world"
result = string1 + " " + string2
print(result)
#Output:  Hello world
```

2. *** (repetition operator):** This operator is used to repeat a string a certain number of times. For example:

```
string = "spam"
result = string * 3
print(result)
# output: spamspamspam
```

3. in and not in (membership operators): These operators are used to check if a string is present in another string. For example:

```
string = "hello world"
print("hello" in string)  # True
print("goodbye" not in string)  # True
```

4. **% (string formatting operator):** This operator is used to format a string with variables. For example:

```
name = "Alice"
age = 30
result = "My name is %s and I'm %d years old." % (name, age)
print(result)
# output: My name is Alice and I'm 30 years old.
```

5. [] (indexing operator): This operator is used to access individual characters in a string. For example:

```
string = "hello"
print(string[0])  # 'h'
print(string[-1])  # 'o'
```

6. Range slicing, denoted by [:] is a way to extract a substring from a string in Python. It works by specifying the start and end indices of the substring that you want to extract. For example, consider the following string:

```
s = "hello world"
```

To extract the substring "hello", you can use the range slicing syntax s[0:5]:

```
substring = s[0:5]
print(substring)
Output:
hello
```

In this example, s[0:5] specifies the range from index 0 (inclusive) to index 5 (exclusive), which corresponds to the substring "hello".

You can also omit the starting or ending index to specify a range that starts from the beginning or ends at the end of the string, respectively.

For example, s[:5] extracts the substring "hello", and s[6:] extracts the substring "world".

```
substring1 = s[:5]
substring2 = s[6:]
print(substring1)
print(substring2)
# output :
hello
world
```

Range slicing can also be used with negative indices, which count from the end of the string. For example, **s[-5:-1]** extracts the substring "worl".

```
substring = s[-5:-1]
print(substring)
# output:
worl
```

Range slicing is a powerful feature in Python strings that allows you to extract substrings quickly and easily.

Format Operators

Format operators are used in print statement. There are two format operators:

1. **%**
2. **format**()

Both % and **format()** are string formatting operators in Python that allow you to insert values into a string.

1. % operator:

Syntax :
print("%s1 %s2 %s3 ------"%(arg1,arg2,arg3,------argn))

Here,
s1,s2 ,.....sn are conversion specifiers like **d**(for int),**f**(for float),**s**(for string) ,etc.
and
arg1,arg2....argn are variables or values.

The % operator is an older way to format strings and is based on C-style string formatting. It uses placeholders in the string, such as %s and %d, to indicate where variables should be inserted. Here's an example:

```
name = "Alice"
age = 30
result = "My name is %s and I'm %d years old." % (name, age)
print(result)
# Output:
My name is Alice and I'm 30 years old.
```

In this example, %s is a placeholder for a string value and %d is a placeholder for an integer value. The variables name and age are inserted into the string using the % operator.

Example :

```
# Example for usage of % operator
x= 19
y= 3.15
z ="Context"
print("The values of x= %d , y = %f and x = %s"%(x,y,z))
```

```
The values of x= 19 , y = 3.150000 and x = Context
```

2. format() function

Syntax :

print('{0} {1} .....{n}'.format(agr1,agr2,agr3....argn)

Here,

0,1,2,------are position specifiers

and

arg1,arg2,-----argn are variables/values

The format() method is a newer way to format strings and is more versatile than the % operator. It uses curly braces {} to indicate where variables should be inserted. Here's an example:

```
name = "Alice"
age = 30
result = "My name is {} and I'm {} years old.".format(name, age)
print(result)
```

 # Output: My name is Alice and I'm 30 years old.

Example :

```
x = 225
y= 19.75
z = "Contextual AI"
print("The values of x ={0}, y= {1} and z = {2}".format(x,y,z))
```

```
The values of x =225, y= 19.75 and z = Contextual AI
```

Sample Programs:

1.**Write a Python Program to Check if the String is Symmetrical or Palindrome**:

A string is said to be symmetrical if both halves of the string are the same, and a string is said to be a palindrome if one half of the string is the opposite of the other half or if the string appears the same whether read forward or backward.

```
instr = input("Enter the string: ")

mid = int(len(instr) / 2)

if len(instr) % 2 == 0:  # even
    first_string = instr[:mid]
    second_string = instr[mid:]
else:  # odd
    first_string = instr[:mid]
    second_string = instr[mid+1:]

# condition to symmetric
if first_string == second_string:
    print(instr, 'Entered string is symmertical')
else:
    print(instr, 'Entered string is not symmertical')

# condition to check palindrome
if first_string == second_string[::-1]:
    print(instr, 'Entered string is palindrome')
else:
    print(instr, 'Entered string is not palindrome')
```

Output 1 :

```
Enter the string: adad
adad Entered string is symmertical
adad Entered string is not palindrome
```

Output 2

```
Enter the string: gadag
gadag Entered string is not symmertical
gadag Entered string is palindrome
```

2. Write a Python Program to find the length of string

```
My_string = "Context Aware Computing"
#Using for loop and in operator
def Length(My_string):
    count = 0
    for i in My_string:
        count += 1
    return count
#Using while loop and Slicing
def Length(My_string):
    count = 0
    while My_string[count:]:
        count += 1
    return count
#Using string methods join and count
def Length(My_string):
    if not My_string:
        return 0
    else:
        some_random_string = 'xyz'
        return ((some_random_string).join(My_string)).count(some_random_string) + 1
#Using the built-in function len
print("length of the String using for loop and in operator:", len(My_string))
print("length of the String Using while loop and Slicing:", Length(My_string))
print("length of the String Using string methods join and count:", Length(My_string))
print("length of the String Using the built-in function len:", Length(My_string))
```

Output :

```
length of the String using for loop and in operator: 23
length of the String Using while loop and Slicing: 23
length of the String Using string methods join and count: 23
length of the String Using the built-in function len: 23
```

3. Write a Python Program to Count the Number of Digits, Alphabets, and Other Characters in a String

```
My_string = "tocxten@2022"

alphabets = 0
digits = 0
special_char =0

# To Count Alphabets, Digits and Special Characters in a String using
# For Loop, isalpha() and isdigit() function
for i in range(len(My_string)):
    if(My_string[i].isalpha()):
        alphabets = alphabets + 1
    elif(My_string[i].isdigit()):
        digits = digits + 1
    else:
        special_char = special_char + 1

print("Total Count of Alphabets in given String:  ", alphabets)
print("Total Count of Digits in given String:  ", digits)
print("Total Count of Special Characters in given String:  ", special_char)
```

Output :

```
Total Count of Alphabets in given String:   7
Total Count of Digits in given String:   4
Total Count of Special Characters in given String:   1
```

4. Write a Python Program to Count the Number of Vowels in a String

```
# Function to check the Vowel
def vowel(chars):
    return chars.upper() in ['A', 'E', 'I', 'O', 'U']

# Returns count of vowels in str
def countofVowels(My_string):
    count = 0
    for i in range(len(My_string)):

        # Check for vowel
        if vowel(My_string[i]):
            count += 1
    return count

My_string = 'Contextual Artificial Intelligence'

# Total number of Vowels
print("Number of vowels in the given string is:",countofVowels(My_string))
```

```
Number of vowels in the given string is: 14
```

5. Write a Python Program to Split and Join a String

```
#using split and join methods
def split_the_string(string):
  # Split the string based on whitespace delimiter
    substring_list = string.split(' ')
    return substring_list

def join_the_string(substring_list):
   # Then join the list of substrings based on '|' delimiter
    string1 = '/'.join(substring_list)
    return string1
# Driver Code
if __name__ == '__main__':
    test_string1 = 'Python is the best'

  # Splitting a string
substring_list = split_the_string(test_string1)
print(substring_list)

# Join list of strings into one
res_string = join_the_string(substring_list)
print(res_string)
```

```
['Python', 'is', 'the', 'best']
Python/is/the/best
```

6. Write a Python program to display a largest word from a string

```
str=input("Enter any String :")
L = str.split()
max=0
b=""
for i in L:
   if len(i) > max:
       b=i
       max=len(i)
print("Longest word is ",b)
```

7. Write a Python program to display unique words from a string

```
str=input("Enter any String :")
L=str.split( )
L1=[ ]
for i in L:
    if i not in L1:
        L1.append(i)
str=""
for i in L1:
    str=str+" "+i
print("String with unique words are :",str)
```

8.**Write a Program to accept a word and display a ASCII value of each character of words**

```
## To print ASCII Value
instring = input("Enter a string")
for i in instring:
    print("ASCII VALUE of ",i," is : ",ord(i) )
```

```
Enter a stringPalguni G T
ASCII VALUE of  P  is :  80
ASCII VALUE of  a  is :  97
ASCII VALUE of  l  is :  108
ASCII VALUE of  g  is :  103
ASCII VALUE of  u  is :  117
ASCII VALUE of  n  is :  110
ASCII VALUE of  i  is :  105
ASCII VALUE of     is :  32
ASCII VALUE of  G  is :  71
ASCII VALUE of     is :  32
ASCII VALUE of  T  is :  84
```

2.3Tuples

In Python, a tuple is an ordered, immutable collection of elements, typically used to group related data together. Tuples are very similar to lists, with the main difference being that tuples are immutable, meaning that once they are created, their elements cannot be modified. Because of this, tuples can be used in situations where data should not be modified after creation, or where it is desirable to protect the data from being accidentally changed. The index value of tuple starts from 0.

Tuples use parentheses (), whereas lists use square brackets [ ].

Examples:

```
tup1 = ('physics', 'chemistry', 1997, 2000)
tup2 = (1, 2, 3, 4, 5 )
tup3 = "a", "b", "c", "d"
```

2.3.1. Creation of Tuples

Creating Empty Tuples:

To create an empty tuple in Python, you can simply use a pair of parentheses with nothing inside:

```
empty_tuple = ( )
```

This creates an empty tuple named empty_tuple. Alternatively, you can use the tuple() constructor function to create an empty tuple:

```
empty_tuple = tuple()
```

This has the same effect as using empty parentheses. Empty tuples can be useful as placeholders for values that will be filled in later, or in cases where you need to pass a tuple to a function but don't have any data to include in it. Note that once a tuple is created, you cannot add or remove elements from it, so an empty tuple will remain empty unless you assign values to it.

Creating tuple with single element

To create a tuple with a single element in Python, you need to be careful because using just parentheses to enclose the element will create a different type of object, a string, integer or float. For example, if you write:

```
my_tuple = (4)
```

The my_tuple variable will not be a tuple but an integer value of 4. To create a tuple with a single element, you can add a trailing comma after the element within parentheses:

```
my_tuple = (4,)
```

This creates a tuple with a single element of value 4. It is important to add the comma to distinguish the tuple from an integer value. Here is another example:

```
my_tuple = ("apple",)
```

This creates a tuple with a single element of value “apple”.

Without the trailing comma, Python will interpret it as a string, like this:

```
my_string = ("apple")
```

In this case, my_string will be a string variable, not a tuple.

Creation of Tuples with Multiple Values

To create a tuple with multiple values in Python, you can separate each element with a comma and enclose them all in parentheses:

```
my_tuple = (1, 2, "three", 4.0)
```

This creates a tuple named my_tuple with four elements: the integers 1 and 2, the string “three”, and the floating-point number 4.0. You can create a tuple with any number of elements, and each element can be of a different type. It is also possible to create a tuple without explicitly using parentheses, as long as there are commas between the values:

```
my_tuple = 1, 2, "three", 4.0
```

This has the same effect as the previous example. You can also use variables to create a tuple:

```
x = 1
y = 2
my_tuple = (x, y)
```

In this case, my_tuple will be a tuple containing the values of x and y, which are both integers. Tuples are flexible data structures that can hold any combination of values, and are often used to group related data together in a way that is easy to work with.

Example :

```
# Creation of tuple with Multiple Values
tup1 = ("Palguni","SEM1","MCE","Hassan")
print("Tuple of Strings : ", tup1)
tup2 = (1,2,3,4,5,6)
print("Tuple of numbers : ", tup2)
tup3 = "A","B","C","D","E"
print("Tuple of Chaacters without using parentheses : ", tup3)
```

```
Tuple of Strings :  ('Palguni', 'SEM1', 'MCE', 'Hassan')
Tuple of numbers :  (1, 2, 3, 4, 5, 6)
Tuple of Chaacters without using parentheses :  ('A', 'B', 'C', 'D', 'E')
```

Example :

```
# Tuples with single character
T =('C','O','N','T','E','X','T')
print(T)
```

```
('C', 'O', 'N', 'T', 'E', 'X', 'T')
```

Example :

```
# Tuple with string values
T = ("Sunday","Monday","Tuesday","Wednesday","Thursday")
print(T)
```

```
('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday')
```

2.3.2. Adding new elements to Tuple

In Python, tuples are immutable data structures, which means that once a tuple is created, you cannot add, remove, or modify its elements. If you need to add a new element to a tuple, you will need to create a new tuple that includes the original tuple's elements as well as the new element.

Here's an example of how to create a new tuple by adding a new element to an existing tuple:

```
original_tuple = (1, 2, 3)
new_tuple = original_tuple + (4,)
```

In this example, we start with an original tuple `(1, 2, 3)`. We create a new tuple `new_tuple` by concatenating the original tuple with a new tuple containing the single element `4` using the `+` operator. Note that we also include a comma after the `4` to ensure that it is interpreted as a tuple with a single element, rather than just a number.

After running this code, new_tuple will contain (1, 2, 3, 4).

It's important to note that creating a new tuple in this way can be inefficient if you are working with very large tuples, since it involves creating a new copy of the entire tuple. If you need to add or remove elements frequently, you might want to consider using a different data structure, such as a list, which allows you to modify its contents.

Example :

```
# Add new element to Tuple
T=(10,20,30,40) +(50,) # This will not modify T
```

```
print(T)
```

```
(10, 20, 30, 40, 50)
```

```
T=(10,20,30,40)
T = T + (50,) # This will  modify T
print(T)
```

```
(10, 20, 30, 40, 50)
```

2.3.3.Accessing Values in Tuples

The values in tuples can be accessed using squared brackets , index number and slicing . You can access individual values within a tuple in Python using indexing. Indexing starts at 0 for the first element of the tuple, 1 for the second element, and so on. For example, given the tuple:

```
my_tuple = ('apple', 'banana', 'cherry', 'date')
```

You can access the first element using:

```
my_tuple[0]    # returns 'apple'
```

The second element can be accessed using:

```
my_tuple[1]    # returns 'banana'
```

And so on.

You can also use negative indexing to access elements from the end of the tuple. The last element can be accessed using:

```
my_tuple[-1]   # returns 'date'
```

The second-last element can be accessed using:

```
my_tuple[-2]   # returns 'cherry'
```

And so on.

If you try to access an index that is out of range, you will get an IndexError. For example, if you try to access `my_tuple[4]` in the example above, you will get an IndexError because the tuple only has four elements.
In addition to indexing, you can also use slicing to access a subset of the elements in a tuple. Slicing allows you to extract a range of elements from the tuple, and returns a new tuple containing those elements.

For example, to get the second and third elements of the `my_tuple` tuple, you can use slicing:

```
my_slice = my_tuple[1:3]    # returns ('banana', 'cherry')
```

In this example, the slice notation `[1:3]` indicates that we want to extract elements starting at index 1 (the second element) up to, but not including, index 3 (the fourth element). The result is a new tuple containing the elements ‘banana’ and ‘cherry’.

Example :

```
T1 =(1,2,3,4,5,6)
print(T1[0])
print(T1[3])
print(T1[0:6])
print(T1[0:6:2])
print(T1[:])
print(T1[::])
print(T1[::-1])
```

```
1
4
(1, 2, 3, 4, 5, 6)
(1, 3, 5)
(1, 2, 3, 4, 5, 6)
(1, 2, 3, 4, 5, 6)
(6, 5, 4, 3, 2, 1)
```

2.3.4. Updating Tuples

In Python, tuples are immutable, which means that once a tuple is created, its contents cannot be modified. This means that you cannot add, remove or update elements in a tuple once it has been created.

However, you can create a new tuple with updated values based on the existing tuple. To do this, you can use slicing and concatenation to extract the parts of the tuple you want to keep, and add or replace the elements you want to update. Here's an example:

```
my_tuple = (1, 2, 3, 4)
new_tuple = my_tuple[:2] + (5,) + my_tuple[3:]
```

In this example, we start with a tuple my_tuple that contains the values (1, 2, 3, 4). We create a new tuple new_tuple by extracting the first two elements of my_tuple using slicing (my_tuple[:2]), adding the value 5 as a new element in a tuple ((5,)), and then concatenating the remaining elements of my_tuple after the fourth element using slicing (my_tuple[3:]).

After running this code, new_tuple will contain the values (1, 2, 5, 4). Note that we are not actually modifying the original tuple, but creating a new tuple with the desired values.

Although tuples are immutable, they are useful data structures because they allow you to group related data together in a way that is easy to work with, and can be passed around in your code without fear of the contents being modified.

Example :

Tuples are immutable and hence cannot change the value of tuple elements . Consider the following example :

```
T1= (12,34.56)
T1[0]= 100
T1[2]= 'abc'
T1[3] = 'xyz'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_15176\2483836842.py in <module>
      1 T1= (12,34.56)
----> 2 T1[0]= 100
      3 T1[2]= 'abc'
      4 T1[3] = 'xyz'

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

In the above example attempt is made to create tuple of the form (100, 34.56, 'abc', 'xyz') by item assignment. Type error is generated because python does not support item assignment

for tuple objects. However a new tuple can be generated using concatenation as illustrated below :

```
T1= (12,34.56)
T2 = ('abc','xyz')
T3 = (100,) + (T1[1],) + T2[:]
print(T3)
```

```
(100, 34.56, 'abc', 'xyz')
```

2.3.5. Deleting Tuple Elements

In Python, tuples are immutable, which means that once a tuple is created, its contents cannot be modified. This means that you cannot delete elements from a tuple. However, you can create a new tuple that contains all the elements of the original tuple except for the one you want to remove.

To remove an element from a tuple, you can use slicing and concatenation to create a new tuple that contains all the elements before the one you want to remove, as well as all the elements after the one you want to remove. Here's an example:

```
my_tuple = (1, 2, 3, 4)
new_tuple = my_tuple[:2] + my_tuple[3:]
```

In this example, we start with a tuple my_tuple that contains the values (1, 2, 3, 4). We create a new tuple new_tuple by extracting the first two elements of my_tuple using slicing (my_tuple[:2]), and concatenating the remaining elements of my_tuple after the third element using slicing (my_tuple[3:]).

After running this code, new_tuple will contain the values (1, 2, 4), which is the original tuple with the third element (3) removed. Note that we are not actually modifying the original tuple, but creating a new tuple with the desired values.
Although tuples are immutable and you cannot delete elements from them, they are still useful data structures because they allow you to group related data together in a way that is easy to work with, and can be passed around in your code without fear of the contents being modified.

Example :

Tuple object does not support item or range of items deletion:

```
T = ('CSE','AI',"DS","ECE")
print(T)
```

```
('CSE', 'AI', 'DS', 'ECE')
```

```
del T[2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_15176\1692257482.py in <module>
----> 1 del T[2]

TypeError: 'tuple' object doesn't support item deletion
```

```
del T[0:2]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_15176\4055602846.py in <module>
----> 1 del T[0:2]

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

However, the entire tuple can be deleted as illustrated below:

```
T = ('CSE','PHY',"QC","CSBS")
print(T)
```

```
('CSE', 'PHY', 'QC', 'CSBS')
```

```
del T
```

```
print(T)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_15176\2596911637.py in <module>
----> 1 print(T)

NameError: name 'T' is not defined
```

2.3.6. Basic Tuple Operations

Here are some basic tuple operations in Python:

1. **Concatenation:** You can concatenate two tuples using the + operator. The result is a new tuple that contains all the elements from both tuples.

```
tup1 = (1, 2, 3)
tup2 = (4, 5, 6)
tup3 = tup1 + tup2
print(tup3)  # Output: (1, 2, 3, 4, 5, 6)
```

2. **Repetition:** You can repeat a tuple multiple times using the * operator. The result is a new tuple that contains the original tuple repeated the specified number of times.

```
tup1 = (1, 2, 3)
tup2 = tup1 * 3
print(tup2)  # Output: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

3. **Membership Test:** You can check if an element is present in a tuple using the in operator. The result is a Boolean value indicating whether the element is present in the tuple.

```
tup1 = (1, 2, 3)
print(2 in tup1)  # Output: True
print(4 in tup1)  # Output: False
```

4. **Length:** You can find the length of a tuple using the len() function. The result is an integer representing the number of elements in the tuple.

```
tup1 = (1, 2, 3)
print(len(tup1))  # Output: 3
```

5. **Index:** You can find the index of an element in a tuple using the index() method. The result is an integer representing the position of the element in the tuple.

```
tup1 = (1, 2, 3)
print(tup1.index(2))  # Output: 1
```

6. **Count:** You can count the number of occurrences of an element in a tuple using the count() method. The result is an integer representing the number of times the element appears in the tuple.

```
tup1 = (1, 2, 3, 2)
print(tup1.count(2))  # Output: 2
```

7. **Iteration** : To iterate over a tuple, you can use a for loop, which will iterate over each element of the tuple in order. Here is an example:

```
my_tuple = (1, 2, 3, 4, 5)

for item in my_tuple:
    print(item)

# Output :
1
2
3
4
5
```

In the above example, we first create a tuple called my_tuple with five elements. We then use a for loop to iterate over each element of the tuple and print it to the console.
You can also use the enumerate() function to iterate over a tuple and get the index of each element. Here is an example:

```
my_tuple = ('apple', 'banana', 'orange')

for index, item in enumerate(my_tuple):
    print(index, item)

# output :
  0 apple
  1 banana
  2 orange
```

In the above example, we use the enumerate() function to iterate over the my_tuple tuple and get the index and value of each element. The enumerate() function returns a tuple with two values: the index of the current item and the item itself. We then use two variables, index and item, to capture these values and print them to the console.

Example : iterating tuple t1

```
t1 = (1,2,3,4,5)
for i in t1:
    print(i,end = ' ')
```

```
1 2 3 4 5
```

Example : iterating tuple

```
for i in (1,2,3,4,5,6,7,8,9,10):
    print(i,end = ' ')
```

```
1 2 3 4 5 6 7 8 9 10
```

2.3.7. Tuple Assignment

Tuple assignment is a feature in Python that allows you to assign multiple variables at once using a single tuple. The syntax for tuple assignment involves putting the values you want to assign in a tuple on the right-hand side of the equals sign (=), and the variables you want to assign them to on the left-hand side of the equals sign. Here is an example:

```
x, y = 3, 4
```

In this example, we are assigning the value 3 to the variable x and the value 4 to the variable y. This is equivalent to writing:

```
x = 3
y = 4
```

Another example:

```
a, b, c = "apple", "banana", "cherry"
```

Here, we are assigning the string “apple” to the variable a, the string “banana” to the variable b, and the string “cherry” to the variable c.

You can also use tuple unpacking to swap the values of two variables, like this:

```
a, b = b, a
```

This code will swap the values of the variables a and b.

Tuple assignment can also be used to assign values from a function that returns a tuple. For example:

```
def rectangle_info(width, height):
    area = width * height
    perimeter = 2 * (width + height)
    return area, perimeter

width = 5
height = 10
area, perimeter = rectangle_info(width, height)

print("Width:", width)
print("Height:", height)
print("Area:", area)
print("Perimeter:", perimeter)
```

In this example, we define a function rectangle_info that takes in the width and height of a rectangle and returns a tuple containing the area and perimeter of the rectangle. We then call the function and use tuple assignment to assign the area and perimeter values to the variables area and perimeter.

Examples on Swapping two numbers / tuples:

Example 1:

```
n1=10
n2=20
print(n1,n2)
```

```
10 20
```

```
temp = n1
n1 = n2
n2 = temp
print(n1,n2)
```

```
20 10
```

Example 2:

```
a = 20
b = 30
print(a,b)
```

```
20 30
```

```
a,b = b,a # swap a and b
print(a,b)
```

```
30 20
```

Example 3:

```
s1 =(1,2,3)
s2 = (4,5,6)
```

```
print(s1)
```

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

```
print(s2)
```

```
(4, 5, 6)
```

The left side is a tuple of variables, while the right side is a tuple of expressions. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. The number of variables on the left and the number of values on the right must be the same:

```
X1 = 2
X2 = 3
X3 = 4
```

```
X1,X2 = X3,X2,X1
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_6028\972330776.py in <module>
----> 1 X1,X2 = X3,X2,X1

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

2.3.8. Tuple Slicing

Slice operator works on Tuple also. This is used to display more than one selected value on the output screen. Slices are treated as boundaries and the result will contain all the elements between boundaries. Tuple slicing is a way to extract a subset of elements from a tuple. Tuple slicing is done using the slicing operator:
The syntax for tuple slicing is as follows:

tuple[start:stop:step]

where start is the index of the first element to be included in the slice, stop is the index of the first element to be excluded from the slice, and step is the stride between elements. Where start, stop & step all three are optional. If we omit first index, slice starts from '0'. On omitting stop, slice will take it to end. Default value of step is 1.

Here are some examples of tuple slicing in Python:

```
# create a tuple
my_tuple = (1, 2, 3, 4, 5)

# get a slice of the first three elements
first_three = my_tuple[0:3]
print(first_three)  # Output: (1, 2, 3)

# get a slice of the last three elements
last_three = my_tuple[-3:]
print(last_three)  # Output: (3, 4, 5)

# get a slice of every other element
every_other = my_tuple[::2]
print(every_other)  # Output: (1, 3, 5)

# reverse the tuple
reverse_tuple = my_tuple[::-1]
print(reverse_tuple)  # Output: (5, 4, 3, 2, 1)
```

In the first example, we get a slice of the first three elements of the tuple my_tuple using the start index 0 and the stop index 3. In the second example, we get a slice of the last three elements of the tuple my_tuple using the negative start index -3. In the third example, we get a slice of every other element of the tuple my_tuple using a step size of 2. In the fourth example, we reverse the order of the tuple my_tuple using a negative step size.

Example :Tuple slicing

```
T=(10,20,30,40,50)
```

```
print(T)
```

```
(10, 20, 30, 40, 50)
```

```
print(T[:])
```

```
(10, 20, 30, 40, 50)
```

```
print(T[1:3])
```

```
(20, 30)
```

```
print(T[0:4:2])
```

```
(10, 30)
```

```
print(T[::-1])
```

```
(50, 40, 30, 20, 10)
```

2.3.9. Comparing Tuples

In Python, tuples are compared element-wise, starting from the first element. The comparison stops as soon as a mismatch is found between two elements or one of the tuples has been fully compared. If all the elements in the tuples are equal, then the tuples are considered equal.

The comparison of tuples is done based on the following rules:

1. If both tuples have the same length, then the comparison is done element-wise until a mismatch is found or all elements have been compared.
2. If the first mismatched element in the tuples is a numeric type (integer, float, complex), then the comparison is done based on the numeric value.
3. If the first mismatched element in the tuples is a string, then the comparison is done based on the ASCII value of the characters.
4. If the first mismatched element in the tuples is a tuple, then the comparison is done recursively.
5. If all elements in the tuples have been compared and are equal, then the tuples are considered equal.

Here are some examples of comparing tuples in Python(Tuples can be compared using comparison operators like <, >, and == as illustrated below) :

```
# comparing tuples with numeric values
tuple1 = (1, 2, 3)
tuple2 = (1, 2, 4)
print(tuple1 < tuple2)  # Output: True

# comparing tuples with string values
tuple3 = ('apple', 'banana')
tuple4 = ('apple', 'cherry')
print(tuple3 < tuple4)  # Output: True

# comparing tuples with nested tuples
tuple5 = (1, 2, (3, 4))
tuple6 = (1, 2, (3, 5))
print(tuple5 < tuple6)  # Output: True
```

In the first example, the tuples `tuple1` and `tuple2` are compared element-wise, and since the third element in `tuple1` is less than the third element in `tuple2`, the result is `True`.
In the second example, the tuples `tuple3` and `tuple4` are compared element-wise, and since the second element in `tuple3` is less than the second element in `tuple4` based on the ASCII value, the result is `True`.
In the third example, the tuples `tuple5` and `tuple6` are compared element-wise, and since the third element in `tuple5` is less than the third element in `tuple6`, the result is `True`.

Example :Comparing Tuples

```
V1 = (10,20,30)
V2 = (100,200,300)
V3 = (10,20,30)
```

```
V1==V2
```

```
False
```

```
V1==V3
```

```
True
```

```
V1>V2
```

```
False
```

```
V1>V3
```

```
False
```

```
V1<V2
```

```
True
```

2.3.10. max( ) and min() functions

In Python, the max() and min() functions can be used to find the maximum and minimum values in a tuple. The max() function returns the largest element in a tuple, while the min() function returns the smallest element in a tuple. Both functions can be used with tuples containing numeric values or with tuples containing strings. However, if a tuple contains a mix of numeric and string values, the max() and min() functions will raise a TypeError exception. Here are some examples of using the max() and min() functions with tuples:

```
# using max() and min() with tuples containing numeric values
tuple1 = (1, 5, 3, 7, 2, 8, 4)
print(max(tuple1))  # Output: 8
print(min(tuple1))  # Output: 1

# using max() and min() with tuples containing strings
tuple2 = ('apple', 'banana', 'cherry', 'date')
print(max(tuple2))  # Output: 'date'
print(min(tuple2))  # Output: 'apple'
```

In the first example, the max() function returns the largest element in tuple1, which is 8. The min() function returns the smallest element in tuple1, which is 1.
In the second example, the max() function returns the element with the largest ASCII value in tuple2, which is 'date'. The min() function returns the element with the smallest ASCII value in tuple2, which is 'apple'.

Note that the max() and min() functions work with any iterable object in Python, not just tuples.

2.4 Dictionaries

A dictionary is a collection of items that are unordered, changeable, and indexed. Dictionary items are made up of Key – value pair and each key is separated from its value by a colon (:) and whole thing is enclosed in curly braces as illustrated in the following syntax:

> **Syntax:** **my_dict** = {'key1': 'value1','key2': 'value2','key3': 'value3'…'keyn': 'valuen'}

Example:

```
my_dict = {
    "brand":"Maruthi",
    "Model": "Suzuki",
    "Year":2023
}
```

```
print(my_dict)
```

```
{'brand': 'Maruthi', 'Model': 'Suzuki', 'Year': 2023}
```

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples. A dictionary is an extremely useful data storage construct for storing and retrieving all key value pairs, where each element is accessed (or indexed) by a unique key. However, dictionary keys are not in sequences and hence maintain no left-to right order.
Dictionary is a mapping between a set of indices (called **keys)** and a set of **values**. Each key maps a value. The association of a key and a value is called a key-value pair.

2.4.1 **Creating Dictionaries:**

2.4.1.1 Creating Empty Dictionary: Empty dictionaries can be created using curly braces and **dict()** function as illustrated below:

```
# Creating Empty Dictionaries
ed1 = {}  # using curly braces
ed2 = dict() # using dict() function
print(ed1)
print(ed2)
```

```
{}
{}
```

2.4.1.2 Creating Dictionary with Multiple Elements: Dictionaries with multiple elements can be created using curly braces. Each element containing key value pair are separated by comma operator as illustrated below:

Example 1:

```
# Creating dictionary with multiple elements
dict1 = {"Name":"Palguni","USN":"4MCECSBS123","Course": "BE","Branch":"CSBS","SEM":"1"}
print(dict1)
```

```
{'Name': 'Palguni', 'USN': '4MCECSBS123', 'Course': 'BE', 'Branch': 'CSBS', 'SEM': '1'}
```

Example 2 :

```
H=dict()
```

```
H["one"]="keyboard"
H["two"]="Mouse"
H["three"]="printer"
H["Four"]="scanner"
print(H)
```

```
{'one': 'keyboard', 'two': 'Mouse', 'three': 'printer', 'Four': 'scanner'}
```

2.4.2 Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example:

```
student = {"Name":"Palguni","USN":"4MCECSBS123","Course": "BE","Branch":"CSBS","SEM":"1"}
print(student["Name"])
print(student["USN"])
print(student["Course"])
print(student["Branch"])
print(student["SEM"])
```

```
Palguni
4MCECSBS123
BE
CSBS
1
```

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows –

```
print(student["Age"])
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8020\985111816.py in <module>
----> 1 print(student["Age"])

KeyError: 'Age'
```

2.4.3 Updating Dictionary

Dictionary can be updated by adding a *new entry or a key-value pair, modifying an existing entry, or deleting an existing entry* as shown below in the simple example:

```
student = {"Name":"Raju","Class":8}
print(student)
```

```
{'Name': 'Raju', 'Class': 8}
```

```
student["Age"] = 10
student["School"] = "Contextual International School"
print(student)
```

```
{'Name': 'Raju', 'Class': 8, 'Age': 10, 'School': 'Contextual International School'}
```

2.4.4 Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. Consider a dictionary **dictx** as illustrated below:

```
dictx = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print(dictx)
```

```
{'Name': 'Zara', 'Age': 7, 'Class': 'First'}
```

Following example illustrates the deletion of single element with key '**Name**':

```
del dictx['Name']; # remove entry with key 'Name'
print(dictx)
```

```
{'Age': 7, 'Class': 'First'}
```

The clear() method can be used to delete all the elements of dictionary as illustrated below:

```
dictx.clear()     # remove all entries in dict
print(dictx)
```

```
{}
```

One can also delete the entire dictionary in a single operation. To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

```
del dictx        # delete entire dictionary
print(dictx)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_8020\86569096.py in <module>
      1 del dictx        # delete entire dictionary
----> 2 print(dictx)

NameError: name 'dictx' is not defined
```

2.4.5 Properties of Dictionary Keys

Dictionary Values have no restrictions and that they can be of any arbitrary object. Keys are immutable type with following properties:

1. **More than one entry per key is not allowed. When duplicate key is allowed the last assignment wins:**

```
dictX = {'Name':"Raju",'Age':16,"Name":"Rahul"}
print(dictX)
```

```
{'Name': 'Rahul', 'Age': 16}
```

2. **Since Keys are immutable, one can use strings , numbers or tuples as dictionary keys.**

2.4.6 Python Dictionary items() method

It returns the content of dictionary as a list of key and value. The key and value pair will be in the form of a tuple, which is not in any particular order.

Syntax: dictionary. items()

Example:

```
D={'1':'Sunday','1':'Monday','3':'Tuesday','4':'Wednesday'}
D.items()
```

```
dict_items([('1', 'Monday'), ('3', 'Tuesday'), ('4', 'Wednesday')])
```

2.4.7 Python Dictionary keys() method

It returns a list of the key values in a dictionary, which is not in any particular order.

Syntax : dictionary.keys()

Example :

```
D={'1':'Sunday','1':'Monday','3':'Tuesday','4':'Wednesday'}
D.keys()
```

```
dict_keys(['1', '3', '4'])
```

2.4.8 Python Dictionary Values Method

It returns a list of values from key-value pairs in a dictionary, which is not in any particular order. However, if we call both the items () and values() method without changing the dictionary's contents between these two (items() and values()), Python guarantees that the order of the two results will be the same.

```
D = {'1':'Rohan','2':'Raju','3':'Pallu','4':'Pallavi','5':'Deeksha'}
D.values()
```

```
dict_values(['Rohan', 'Raju', 'Pallu', 'Pallavi', 'Deeksha'])
```

```
D.items()
```

```
dict_items([('1', 'Rohan'), ('2', 'Raju'), ('3', 'Pallu'), ('4', 'Pallavi'), ('5', 'Deeksha')])
```

2.4.9 Traversing Dictionary elements using for loop

In Python, you can traverse the elements of a dictionary using various methods. Here are a few common ways to iterate over dictionary elements.

Example 1: Traversing Dictionary using keys

```
D = {'1':'Rohan','2':'Raju','3':'Pallu','4':'Pallavi','5':'Deeksha'}
for i in D.keys():
    print(i,":",D[i])
```

```
1 : Rohan
2 : Raju
3 : Pallu
4 : Pallavi
5 : Deeksha
```

Example 2 : Traversing Dictionary using values

```
D = {'1':'Rohan','2':'Raju','3':'Pallu','4':'Pallavi','5':'Deeksha'}
for i in D.values():
    print(i)
```

```
Rohan
Raju
Pallu
Pallavi
Deeksha
```

Example 3 : Traversing Dictionary using values

```
D = {'1':'Rohan','2':'Raju','3':'Pallu','4':'Pallavi','5':'Deeksha'}
for i in D.items():
    print(i)
```

```
('1', 'Rohan')
('2', 'Raju')
('3', 'Pallu')
('4', 'Pallavi')
('5', 'Deeksha')
```

2.4.10 Membership Operators with Dictionary

Membership operators (in and not in) can be used to check for the presence or absence of keys in a dictionary. Here's how you can use membership operators with dictionaries in Python:

Example 1: Checking whether a key or value exists in a dictionary

```
D = {'1':'Rohan','2':'Raju','3':'Pallu','4':'Pallavi','5':'Deeksha'}
"10" in D.keys()
```

```
False
```

```
D = {'1':'Rohan','2':'Raju','3':'Pallu','4':'Pallavi','5':'Deeksha'}
"2" in D.keys()
```

```
True
```

```
D = {'1':'Rohan','2':'Raju','3':'Pallu','4':'Pallavi','5':'Deeksha'}
"Pallu" in D.values()
```

```
True
```

Example 2: Check if a key does not exist in the dictionary

```
if "key4" not in my_dict:
   print("Key 'key4' does not exist in the dictionary.")
```

2.4.11 Comparing Dictionaries

To compare dictionaries in Python, you can use the equality operator (==) or the cmp() function. Here's how you can compare dictionaries using these methods:

1. **Using the equality operator (==):**

```
dict1 = {"key1": "value1", "key2": "value2"}
dict2 = {"key1": "value1", "key2": "value2"}

if dict1 == dict2:
   print("The dictionaries are equal.")
else:
   print("The dictionaries are not equal.")
```

2. Using the cmp( ) function (available in Python 2):

```
dict1 = {"key1": "value1", "key2": "value2"}
dict2 = {"key1": "value1", "key2": "value2"}

if cmp(dict1, dict2) == 0:
   print("The dictionaries are equal.")
else:
   print("The dictionaries are not equal.")
```

2.4.12 Merging dictionaries: An update ( )

Syntax: Dic_name1.update (dic_name2)

In Python, you can merge dictionaries using the update() method. The update() method modifies the dictionary in place by adding key-value pairs from another dictionary. Here's how you can use the update() method to merge dictionaries:

```
dict1 = {"key1": "value1", "key2": "value2"}
dict2 = {"key3": "value3", "key4": "value4"}
dict1.update(dict2)
print(dict1)
# output: {"key1": "value1", "key2": "value2", "key3": "value3", "key4": "value4"}
```

In the example above, dict1.update(dict2) merges dict2 into dict1. The update() method adds all the key-value pairs from dict2 to dict1. If there are common keys between the dictionaries, the values from dict2 will overwrite the values in dict1 for those keys.

2.4.13 len ()

This method returns the number of key-value pairs in the given dictionary.

```
D = {'1':'Rohan','2':'Raju','3':'Pallu','4':'Pallavi','5':'Deeksha'}
len(D)
```

```
5
```

2.4.14 get() and setdefault() methods with dictionary

1. **get() method:** The get() method retrieves the value associated with a given key in the dictionary. It takes the key as the first argument and an optional second argument specifying a default value to return if the key is not found in the dictionary. Consider the following example:

   my_dict = {"key1": "value1", "key2": "value2"}
   value1 = my_dict.get("key1")
   print(value1) # Output: "value1"
   value3 = my_dict.get("key3", "default")
   print(value3) **# Output:** "default" (since "key3" doesn't exist)

   In the example above, my_dict.get("key1") returns the value associated with "key1", which is "value1". If the key "key3" doesn't exist in the dictionary, my_dict.get("key3", "default") returns the default value "default".

   Another Example:
   picnic_items = {'apples':5,'cups':2}
   'I am bringing ' + str(picnic_items.get('cups',0)) + ' cups'
   #output: 'I am bringing 2 cups'

   'I am bringing ' + str(picnic_items.get('oranges',2)) + ' oranges'
   #Output:'I am bringing 2 oranges'

2. **setdefault() method** : The setdefault() method returns the value associated with a given key in the dictionary. If the key doesn't exist, it adds the key with a default value to the dictionary.

   my_dict = {"key1": "value1", "key2": "value2"}

   value1 = my_dict.setdefault("key1", "default")
   print(value1) # **Output:** "value1"

   value3 = my_dict.setdefault("key3", "default")
   print(value3) # **Output:** "default"

   print(my_dict) # **Output:** {"key1": "value1", "key2": "value2", "key3": "default"}

Example : Get the value of the "color" item, if the "color" item does not exist, insert "color" with the value "**silksilver**":

```
car = {
  "brand": "Hundai",
  "model": "Creta",
  "year": 2020
}
```

```
x = car.setdefault("color", "silksilver")
print(x)
```

```
silksilver
```

```
x = car.setdefault("brand", "Maruthi")
print(x)
```

```
Hundai
```

```
x = car.setdefault("Price",100000)
print(car)
```

```
{'brand': 'Hundai', 'model': 'Creta', 'year': 2020, 'color': 'silksilver', 'Price': 100000}
```

2.4.15 **Converting a dictionary to a list of tuples**

Dictionary have a method called **items()** that returns a list of tuples where each tuple is a key value pair.

```
D = {'1':'Rohan','3':'Raju','4':'Pallu','2':'Pallavi','5':'Deeksha'}
t = list(D.items())
print(t)
```

```
[('1', 'Rohan'), ('3', 'Raju'), ('4', 'Pallu'), ('2', 'Pallavi'), ('5', 'Deeksha')]
```

```
t.sort()
print(t)
```

```
[('1', 'Rohan'), ('2', 'Pallavi'), ('3', 'Raju'), ('4', 'Pallu'), ('5', 'Deeksha')]
```

2.4.16 **Pretty Printing**

In Python, the pprint module provides two functions for pretty-printing data structures: pprint() and pformat(). These functions are useful when you want to display complex data structures, such as dictionaries or lists, in a more readable and structured format. Here's an explanation of pprint.pprint() and pprint.pformat() with examples:

1. **pprint() function:**

   The pprint() function in the pprint module prints a data structure in a formatted and more readable manner. It accepts a single argument, which can be any data structure, such as a dictionary, list, or tuple.

   # printing dictionary elements using print
   student_dict = {'Name':'Tyson',
   'class':'XII',
   'Address': {'Flat':1308,'Block':'A','Lane':2,'City':"Hyderbad"}}
   print(student_dict)
   #output:{'Name': 'Tyson', 'class': 'XII', 'Address': {'Flat': 1308, 'Block': 'A', 'Lane': 2, 'City': 'Hyderbad'}}

   # printing dictionary elements using pprint.pprint()
   import pprint
   pprint.pprint(student_dict)
   {'Address': {'Block': 'A', 'City': 'Hyderbad', 'Flat': 1308, 'Lane': 2},
   'Name': 'Tyson',
   'class': 'XII'}

   # printing dictionary elements using pprint.pprint() with width and indent
   pprint.pprint(student_dict,width = 2, indent =2)

   { 'Address': { 'Block': 'A',

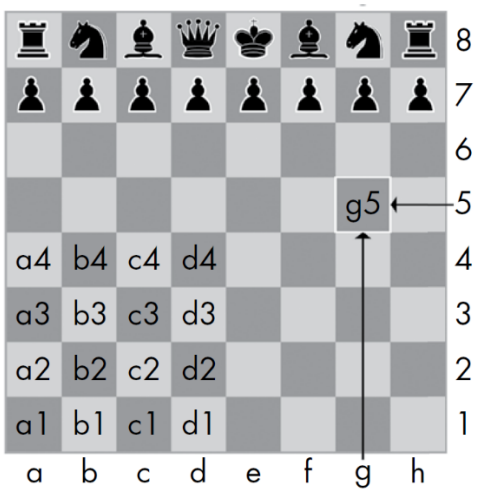
```
            'City': 'Hyderbad',
            'Flat': 1308,
            'Lane': 2},
 'Name': 'Tyson',
 'class': 'XII'}
```

2. **pformat() function:** The pformat() function in the pprint module returns a string representation of the data structure instead of printing it directly. This can be useful if you want to store or manipulate the formatted output.

```
x = pprint.pformat(student_dict,indent =5)
print(x)
#output
{    'Address': {'Block': 'A', 'City': 'Hyderbad', 'Flat': 1308, 'Lane': 2},
     'Name': 'Tyson',
     'class': 'XII'}
```

2.4.17: Using Data Structures to Model Real-World Things

1. Using Data Structures to Model Chess: In *algebraic chess notation*, the spaces on the chessboard are identified by a number and letter coordinate, as in Figure below:



The chess pieces are identified by letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move uses the letter of the piece and the coordinates of its destination. A pair of these moves describes what happens in a single turn (with white going first); for instance, the notation ***2. Nf3 Nc6*** indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game.

Algebraic chess notation is a standard notation system used to record chess moves. It allows players to describe moves using a combination of letters and numbers that represent the squares on the chessboard. Here are the key elements of algebraic chess notation:

Piece Abbreviations:

K: King
Q: Queen
R: Rook
B: Bishop
N: Knight
No abbreviation for pawns

Square Designation:

Each square on the chessboard is designated by a combination of a letter (column) and a number (row). The letters a-h represent the columns from left to right, and the numbers 1-8 represent the rows from bottom to top.

Move Representation:

A move in algebraic notation typically consists of the piece abbreviation (if applicable), the destination square, and additional symbols to indicate the move type:

Pawn moves: Only the destination square is mentioned. If it's a capture, the source file is indicated as well.
Example: e4, exd5 (e4 pawn moves to e5, capturing d5 pawn)
Other pieces: The piece abbreviation is followed by the destination square. If multiple pieces of the same type can move to the same square, the source file or rank is mentioned to disambiguate.
Example: Nf3 (Knight moves to f3), R1e5 (Rook on the first rank moves to e5)

Special Moves:

Castling: O-O for kingside castling and O-O-O for queenside castling.
En Passant: If a pawn captures en passant, the destination square is indicated and "e.p." is added.
Example: exd6 e.p. (e5 pawn captures d6 pawn en passant)

Check and Checkmate:

"+" is added after a move to indicate a check.
"#" is added after a move to indicate a checkmate.

Example: Qh5+ (Queen moves to h5, giving a check), Qh5# (Queen moves to h5, giving a checkmate)

Other Symbols:

"=" is used to indicate pawn promotion. It is followed by the piece abbreviation to which the pawn promotes.

Example: e8=Q (Pawn on e8 promotes to a Queen)

By using algebraic chess notation, players can easily record and communicate their moves. It is also commonly used in chess literature, annotations, and online platforms for game analysis.

2. Using Data Structures to Model Tic-Tac-Toe: A tic-tac-toe board looks like a large hash symbol (#) with nine slots that can each contain an *X*, an *O*, or a blank. To represent the board with a dictionary, you can assign each slot a string-value key, as shown in Figure :

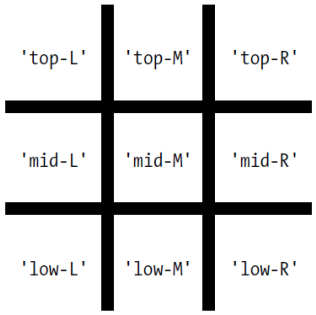


Fig: *slots of a tic-tactoe board with their corresponding keys*

You can use string values to represent what's in each slot on the board: 'X', 'O', or ' ' (a space character). Thus, you'll need to store nine strings. You can use a dictionary of values for this. The string value with the key 'top-R' can represent the top-right corner, the string value with the key 'low-L' can represent the bottom-left corner, the string value with the key 'mid-M' can represent the middle, and so on. This dictionary is a data structure that represents a tic-tac-toe board. Store this board-as-a-dictionary in a variable named theBoard.

theBoard = {
'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
'low-L': ' ', 'low-M': ' ', 'low-R': ' '
}

The data structure stored in the theBoard variable represents the tic-tac-toe board in Figure below:

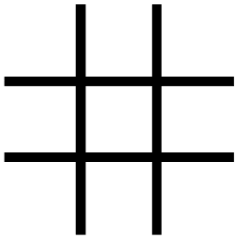


Fig : Empty Tic Tac Toe Board

Since the value for every key in theBoard is a single-space string, this dictionary represents a completely clear board. If player X went first and chose the middle space, you could represent that board with this dictionary:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}
```

The data structure in theBoard now represents the tic-tac-toe board in Figure below:

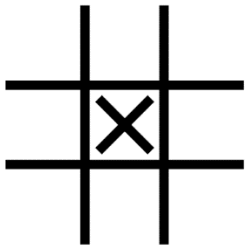


Fig: The First Move

A board where player O has won by placing Os across the top might look like this:

```
theBoard = {'top-L': 'O', 'top-M': 'O', 'top-R': 'O',
            'mid-L': 'X', 'mid-M': 'X', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': 'X'}
```

The data structure in theBoard now represents the tic-tac-toe board in Figure below:

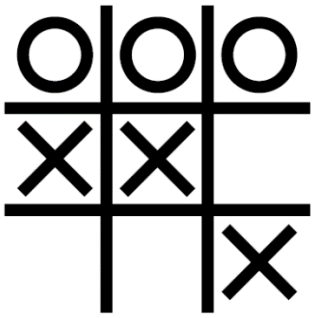


Fig: Player O wins

Python Program to simulate the working of Tic-Tac-Toe:

```
theBoard = {'top-L': ' ', 'top-M': ' ', 'top-R': ' ',
            'mid-L': ' ', 'mid-M': ' ', 'mid-R': ' ',
            'low-L': ' ', 'low-M': ' ', 'low-R': ' '}

def printBoard(board):
    print(board['top-L'] + '|' + board['top-M'] + '|' + board['top-R'])
    print('-+-+-')
    print(board['mid-L'] + '|' + board['mid-M'] + '|' + board['mid-R'])
    print('-+-+-')
    print(board['low-L'] + '|' + board['low-M'] + '|' + board['low-R'])

turn = 'X'

for i in range(9):
    printBoard(theBoard)
    print('Turn for ' + turn + '. Move on which space?')
    move = input()
    theBoard[move] = turn
    if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
printBoard(theBoard)
```

 # OutPut :

```
| |
-+-+-
| |
-+-+-
| |
Turn for X. Move on which space?
mid-M
| |
-+-+-
|X|
-+-+-
| |
Turn for O. Move on which space?
low-L
| |
-+-+-
|X|
-+-+-
```

```
O| |
--snip--
O|O|X
-+-+-
X|X|O
-+-+-
O| |X
Turn for X. Move on which space?
low-M
O|O|X
-+-+-
X|X|O
-+-+-
O|X|X
```

This isn't a complete tic-tac-toe game—for instance, it doesn't ever check whether a player has won—but it's enough to see how data structures can be used in programs.

2.4.18: Nested Dictionaries and Lists

1. Nested Dictionaries: A nested dictionary is a dictionary that contains other dictionaries as its values. It allows you to create a hierarchical structure to organize and store data. Each inner dictionary can have its own set of key-value pairs. Here's an example:

```
student = {
    'name': 'John',
    'age': 20,
    'grades': {
        'math': 90,
        'science': 85,
        'history': 95
    }
}
```

In the example above, the student dictionary has three key-value pairs. The 'grades' key has a value of another dictionary, which contains subject-grade pairs.

You can access the nested values by chaining the keys together. For example, to access the math grade, you can use student['grades']['math'], which will return 90.

2. Nested Lists: A nested list is a list that contains other lists as its elements. It allows you to create a multidimensional structure to store data. Each inner list can have its own set of elements. Here's an example:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

In the example above, the matrix list contains three inner lists, forming a 3x3 matrix. You can access the elements of the nested list using indexing. For example, to access the element 5, you can use matrix[1][1], which corresponds to the second row and second column. Nested lists can be useful for representing grids, matrices, or any other data structures that require multiple dimensions.

Nested dictionaries and lists can be combined to create more complex data structures. For example, you can have a list of dictionaries, where each dictionary represents an entity with multiple properties, or you can have a dictionary that contains lists as its values, allowing you to group related items together.

Remember that when working with nested data structures, you need to carefully consider the structure and access patterns to ensure efficient and correct data manipulation.

Sample Question Bank

1. What is list ? Explain the concepts of list slicing with example.
2. Explain references with example
3. Explain why lists are called Mutable.
4. Discuss the following List operations and functions with examples :
    1. Accessing ,Traversing and Slicing the List Elements
    2. *+ (concatenation) and * (Repetition)*
    3. *append , extend , sort , remove and delete*
    4. *split and join*
5. What is Dictionary ? List merits of Dictionaries over List.
6. Write a python program to accept USN and marks obtained, find maximum ,minimum and students USN who have scored in the range 100 -85, 85-75, 75-60 and 60 marks separately.
7. Define String? Explain at least 5 String functions with examples.
8. Write a python program to display presence of given substring in main string.
9. Define Tuple , List , Strings and Dictionary in Python.
10. Compare tuples with Lists. Explain how tuple can be converted into list and list into tuples.
11. Predict the out put and justify your answer: (i) -11%9 (ii) 7.7//7 (iii) (200 – 70)*10/5 (iv) not “False” (v) 5*|**2
12. What is dictionary ? How it is different from list ? Write a program to count the number of occurrences of character in a string.
13. List any six methods associated with string and explain each of them with example.
14. Write a Python program to swap cases of a given string .
    Input : Java
    Output : jAVA
15. What is Dictionary in Python ? How is it different from List Data type ? Explain how a for loop can be used to traverse the keys of the Dictionary with an example.
16. Discuss the following Dictionary methods in Python with example.
    a. get()
    b. items()
    c. keys()
    d. values()
17. Explain the various string methods for the following operations with examples.
    a. Removing white space characters from the beginning end or both sides of a string
    b. To right -justify , left justify and center a string
18. Explain the methods of list data type in Python for the following operations with suitable code snippets for each.
    - Adding values to a list
    - Removing values from a list
    - Finding a value in a list
    - Sorting the values in a list
19. Write a Python program that accepts a sentence and find the number of words , digits , uppercase letters and lowercase letters
20. Explain the various string methods for the following operations with examples.
    1. Removing white space characters from the beginning, end or both sides of a string
    2. To right -justify, left justify and center a string.

21. What is list ? Explain the concept of slicing and indexing with proper examples.
22. What are the different methods supports in python List . Illustrate all the methods with an example.
23. What is dictionary ? Illustrate with an example python program the usage of nested dictionary.
24. List out all the useful string methods which supports in python . Explain with an example for each method.
25. What are the different steps in project adding Bullets to Wiki Markup.
26. Explain negative indexing, slicing, index (), append(), remove, pop(), insert() and sort() with suitable example in lists.
27. Explain the use of in and not in operators in list with suitable examples.
28. Illustrate with examples, why List are mutable, and strings are immutable.
29. Develop a program to read the student details like Name, USN, and Marks in three subjects. Display the student details, total marks and percentage with suitable messages
30. Develop a program to read the name and year of birth of a person. Display whether the person is a senior citizen or not.
31. Develop a program to generate Fibonacci sequence of length (N). Read N from the console.
32. Write a function to calculate factorial of a number. Develop a program to compute binomial coefficient (Given N and R).
33. Read N numbers from the console and create a list. Develop a program to print mean, variance and standard deviation with suitable messages.
34. Read a multi-digit number (as chars) from the console. Develop a program to print the frequency of each digit with suitable message.
35. Develop a program to calculate the area of rectangle and triangle print the result.
36. Write a function that computes and returns addition, subtraction, multiplication, division of two integers. Take input from user.