NextGen Academy

Towards fulfilling a million dreams

Python Programming

Module II : String, collection lists and Tuples

June 20, 2023

Contents

This unit covers

- Declaring and working with strings
- String manipulation using various string functions
- Introduction to collection lists and their manipulation
- Understanding and manipulating collections lists
- Introduction to tuples and their significance
- Manipulating tuples for various operations

1 Declaring strings

Python is a versatile and widely used programming language that provides various data types to deal with different data structures. One of the most commonly used data types is the string, which represents a sequence of characters. In this blog post, we will explore **how to create a string in Python** using various methods and examples.

1.1 Strings in Python

A string in Python is an ordered sequence of characters enclosed within single or double quotes. It can contain letters, numbers, special characters, and even escape sequences. Strings are immutable in Python, meaning their values cannot be changed after they are created. However, you can create new strings using existing ones.

1.2 Create a string in Python

There are different ways, we can create strings in Python.

1.2.1 Create a string in Python using Single quotes

Creating a string in Python using single quotes is one of the most basic and commonly used methods. You can use single quotes to define a string that contains characters, digits, and special symbols.

Here are some examples and specific scenarios where single quotes are useful:

- You can create a simple string using single quotes as follows:

stringName='United States of America'

print(stringName)

You can see the output:

- Multi-Line Strings with Single Quotes:

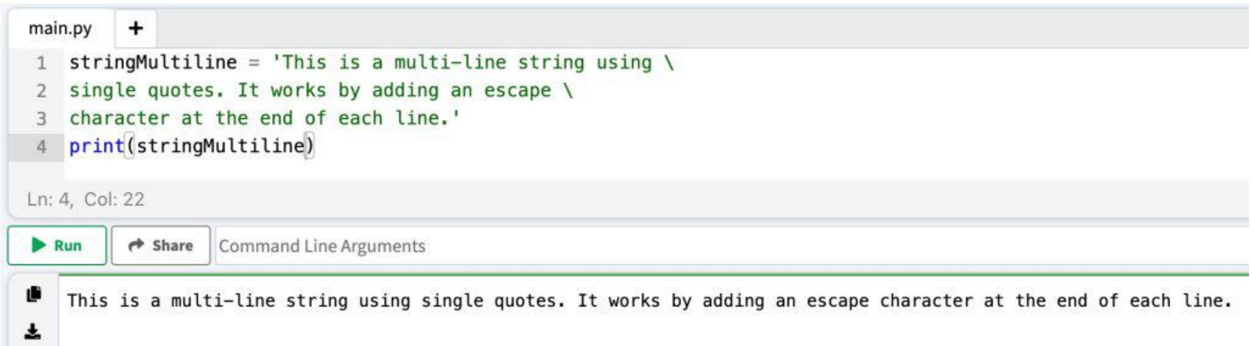
Although single quotes are not meant for defining multi-line strings, you can use an escape character at the end of each line to achieve this in Python.

*stringMultiline = 'This is a multi-line string using *

*single quotes. It works by adding an escape *

character at the end of each line.'

print(stringMultiline)



- String Interpolation with Single Quotes: You can also use single quotes when performing string interpolation using f-strings or the str.format() method:

name = 'John'

age = 30

f_string = f'{name} is {age} years old.'

format_string = '{name} is {age} years old.'.format(name=name, age=age)

print(f_string)

*# **Output:***

John is 30 years old.

print(format_string)

*# **Output:***

John is 30 years old.

This is how we can create a string using single quotes.

1.2.2 Create a string in Python using double quotes

Creating a string in Python using double quotes is another basic and widely used method. Double quotes can be employed to define strings that contain characters, digits, and special symbols.

Here are some examples and specific scenarios where double quotes we are using to create a string in Python.

- Create a string using double quotes:

You can create a simple string using double quotes as follows:

string1 = "New York City, New York"

print(string1)

*# **Output:***

New York City, New York

- Strings with Single Quotes: When your string contains single quotes, you can use double quotes to avoid the need for escaping:

string2 = "It's always sunny in Philadelphia, Pennsylvania"

print(string2)

*# **Output:***

It's always sunny in Philadelphia, Pennsylvania

- Escaping Double Quotes within Double-Quoted Strings:

If your Python string contains double quotes, you'll need to escape them using a backslash (\). Without the escape character, the interpreter would assume that the double quote marks the end of the string:

string3 = "The \"City of Angels\": Los Angeles, California"

print(string3)

*# **Output:***

The "City of Angels": Los Angeles, California

- Combining Double-Quoted Strings to one string

You can concatenate strings defined with double quotes using the + operator in Python:

string4 = "Miami, " + "Florida"

print(string4)

 # Output:

Miami, Florida

- Multi-Line Strings with Double Quotes:

Similar to single quotes, you can use an escape character at the end of each line to create a multi-line string with double quotes in Python:

```
string5 = "Famous US cities include: \
\"New York City, New York\", \
\"Los Angeles, California\", and \
\"Chicago, Illinois\"."
print(string5)
```

This is how to create a string using double quotes in Python.

1.2.3 Create a string in Python using triple quotes(' " " ')

Triple quotes in Python allow you to create strings that span multiple lines or contain both single and double quotes. You can use triple single quotes (''' ''') or triple double quotes (""" """). Here are two examples featuring various city names from the United States of America:

Example 1 – Multi-line String:

```
stringMultiline = """Some popular cities in the United States of America:
- New York City, New York
- Los Angeles, California
- Chicago, Illinois
- Houston, Texas
- Phoenix, Arizona
"""

print(stringMultiline)
```

You can see the output like below:

```python
stringMultiline = """Some popular cities in the United States of America:
- New York City, New York
- Los Angeles, California
- Chicago, Illinois
- Houston, Texas
- Phoenix, Arizona
"""

print(stringMultiline)
```

Ln: 9, Col: 22

Run Share Command Line Arguments

```
Some popular cities in the United States of America:
- New York City, New York
- Los Angeles, California
- Chicago, Illinois
- Houston, Texas
- Phoenix, Arizona
```

1.3 String Operators

1.3.1 The + Operator

The + operator concatenates strings. It returns a string consisting of the operands joined together, as shown here:

```
>>> s = 'foo'
>>> t = 'bar'
>>> u = 'baz'
>>> s + t
'foobar'
>>> s + t + u
'foobarbaz'
>>> print('Go team' + '!!!')
Go team!!!
```

1.3.2 The * Operator

The * operator creates multiple copies of a string. If s is a string and n is an integer, either of the following expressions returns a string consisting of nconcatenated copies of s:

s * n

n * s

Here are examples of both forms:

>>> s = 'foo.'

*>>> s * 4*

'foo.foo.foo.foo.'

*>>> 4 * s*

'foo.foo.foo.foo.'

The multiplier operand n must be an integer. You'd think it would be required to be a positive integer, but amusingly, it can be zero or negative, in which case the result is an empty string:

*>>> 'foo' * -8*

''

If you were to create a string variable and initialize it to the empty string by assigning it the value 'foo' * -8, anyone would rightly think you were a bit daft. But it would work.

The in Operator

Python also provides a membership operator that can be used with strings. The in operator returns True if the first operand is contained within the second, and False otherwise:

>>> s = 'foo'

>>> s in 'That\'s food for thought.'

True

>>> s in 'That\'s good for now.'

False

There is also a not in operator, which does the opposite:

>>> 'z' not in 'abc'

True

>>> 'z' not in 'xyz'

False

1.4 String Manipulation using string functions

Built-in String Functions

As you saw in the tutorial on Basic Data Types in Python, Python provides many functions that are built-in to the interpreter and always available. **Here are a few that work with strings:.**

Function	Description
chr()	Converts an integer to a character
ord()	Converts a character to an integer
len()	Returns the length of a string
str()	Returns a string representation of an object

These are explored more fully below.

ord(c)

Returns an integer value for the given character.

At the most basic level, computers store all information as numbers. To represent character data, a translation scheme is used which maps each character to its representative number.

The simplest scheme in common use is called ASCII. It covers the common Latin characters you are probably most accustomed to working with. **For these characters, ord(c) returns the ASCII value for character c:**

>>> ord('a')

97

>>> ord('#')

35

ASCII is fine as far as it goes. But there are many different languages in use in the world and countless symbols and glyphs that appear in digital media. The full set of characters that potentially may need to be represented in computer code far surpasses the ordinary Latin letters, numbers, and symbols you usually see.

Unicode is an ambitious standard that attempts to provide a numeric code for every possible character, in every possible language, on every possible platform. Python 3 supports Unicode extensively, including allowing Unicode characters within strings.

As long as you stay in the domain of the common characters, there is little practical difference between ASCII and Unicode. But the ord() function will return numeric values for Unicode characters as well:

>>> ord('€')

8364

>>> ord('')

8721

1.4.1 chr(n)

Returns a character value for the given integer.

chr() does the reverse of ord(). Given a numeric value n, chr(n) returns a string representing the character that corresponds to n:

>>> chr(97)

'a'

>>> chr(35)

'#'

chr() handles Unicode characters as well:

>>> chr(8364)

'€'

>>> chr(8721)

''

1.4.2 len(s)

Returns the length of a string.

With len(), you can check Python string length. len(s) returns the number of characters in s:

>>> s = 'I am a string.'

>>> len(s)

14

1.4.3 str(obj)

Returns a string representation of an object.

Virtually any object in Python can be rendered as a string. str(obj) returns the string representation of object obj:

>>> str(49.2)

'49.2'

>>> str(3+4j)

'(3+4j)'

>>> str(3 + 29)

'32'

```
>>> str('foo')
'foo'
```

1.4.4 String Indexing

Often in programming languages, individual items in an ordered set of data can be accessed directly using a numeric index or key value. This process is referred to as indexing.

In Python, strings are ordered sequences of character data, and thus can be indexed in this way. Individual characters in a string can be accessed by specifying the string name followed by a number in square brackets ([]).

String indexing in Python is zero-based: the first character in the string has index 0, the next has index 1, and so on. The index of the last character will be the length of the string minus one.

For example, a schematic diagram of the indices of the string 'foobar' would look like this:

String Indices

The individual characters can be accessed by index as follows:

```
>>> s = 'foobar'

>>> s[0]
'f'
>>> s[1]
'o'
>>> s[3]
'b'
>>> len(s)
6
>>> s[len(s)-1]
'r'
```

Attempting to index beyond the end of the string results in an error:

```
>>> s[6]
Traceback (most recent call last):
File "<pyshell#17>", line 1, in <module>
s[6]
IndexError: string index out of range
```

String indices can also be specified with negative numbers, in which case indexing occurs from the end of the string backward: -1 refers to the last character, -2 the second-to-last character, and so on. Here is the same diagram showing both the positive and negative indices into the string 'foobar':

Positive and Negative String Indices

Here are some examples of negative indexing:

```
>>> s = 'foobar'

>>> s[-1]
'r'
>>> s[-2]
'a'
>>> len(s)
6
>>> s[-len(s)]
'f'
```

Attempting to index with negative numbers beyond the start of the string results in an error:

```
>>> s[-7]
Traceback (most recent call last):
File "<pyshell#26>", line 1, in <module>
s[-7]
IndexError: string index out of range
```

For any non-empty string s, s[len(s)-1] and s[-1] both return the last character. There isn't any index that makes sense for an empty string.

1.4.5 String Slicing

Python also allows a form of indexing syntax that extracts substrings from a string, known as string slicing. If s is a string, an expression of the form s[m:n] returns the portion of s starting with position m, and up to but not including position n:

```
>>> s = 'foobar'
>>> s[2:5]
'oba'
```

Remember: String indices are zero-based. The first character in a string has index 0. This applies to both standard indexing and slicing.

Again, the second index specifies the first character that is not included in the result—the character 'r' (s[5]) in the example above. That may seem slightly unintuitive, but it produces this result which makes sense: the expression s[m:n] will return a substring that is n - m characters in length, in this case, 5 - 2 = 3.

If you omit the first index, the slice starts at the beginning of the string. Thus, s[:m] and s[0:m] are equivalent:

```
>>> s = 'foobar'

>>> s[:4]
'foob'
>>> s[0:4]
'foob'
```

Similarly, if you omit the second index as in s[n:], the slice extends from the first index through the end of the string. This is a nice, concise alternative to the more cumbersome s[n:len(s)]:

```
>>> s = 'foobar'

>>> s[2:]
'obar'
>>> s[2:len(s)]
'obar'
```

For any string s and any integer n (0 n len(s)), s[:n] + s[n:] will be equal to s:

```
>>> s = 'foobar'

>>> s[:4] + s[4:]
'foobar'
>>> s[:4] + s[4:] == s
True
```

Omitting both indices returns the original string, in its entirety. Literally. It's not a copy, it's a reference to the original string:

```
>>> s = 'foobar'
>>> t = s[:]
>>> id(s)
59598496
>>> id(t)
59598496
>>> s is t
True
```

If the first index in a slice is greater than or equal to the second index, Python returns an empty string. This is yet another obfuscated way to generate an empty string, in case you were looking for one:

```
>>> s[2:2]
''
```

```
>>> s[4:2]
''
```

Negative indices can be used with slicing as well. -1 refers to the last character, -2 the second-to-last, and so on, just as with simple indexing. The diagram below shows how to slice the substring 'oob' from the string 'foobar' using both positive and negative indices:

String Slicing with Positive and Negative Indices

Here is the corresponding Python code:

```
>>> s = 'foobar'

>>> s[-5:-2]
'oob'
>>> s[1:4]
'oob'
>>> s[-5:-2] == s[1:4]
True
```

1.4.6 Specifying a Stride in a String Slice

There is one more variant of the slicing syntax to discuss. Adding an additional : and a third index designates a stride (also called a step), which indicates how many characters to jump after retrieving each character in the slice.

For example, for the string 'foobar', the slice 0:6:2 starts with the first character and ends with the last character (the whole string), and every second character is skipped. This is shown in the following diagram:

String Indexing with Stride

Similarly, 1:6:2 specifies a slice starting with the second character (index 1) and ending with the last character, and again the stride value 2 causes every other character to be skipped:

Another String Indexing with Stride

The illustrative REPL code is shown here:

```
>>> s = 'foobar'

>>> s[0:6:2]
'foa'

>>> s[1:6:2]
'obr'
```

As with any slicing, the first and second indices can be omitted, and default to the first and last characters respectively:

```
>>> s = '12345' * 5
>>> s
'1234512345123451234512345'
>>> s[::5]
'11111'
>>> s[4::5]
'55555'
```

You can specify a negative stride value as well, in which case Python steps backward through the string. In that case, the starting/first index should be greater than the ending/second index:

```
>>> s = 'foobar'
>>> s[5:0:-2]
'rbo'
```

In the above example, 5:0:-2 means "start at the last character and step backward by 2, up to but not including the first character."

When you are stepping backward, if the first and second indices are omitted, the defaults are reversed in an intuitive way: the first index defaults to the end of the string, and the second index defaults to the beginning. **Here is an example:**

```
>>> s = '12345' * 5
>>> s
'1234512345123451234512345'
>>> s[::-5]
'55555'
```

This is a common paradigm for reversing a string:

```
>>> s = 'If Comrade Napoleon says it, it must be right.'
>>> s[::-1]
'.thgir eb tsum ti ,ti syas noelopaN edarmoC fI'
```

1.4.7 Interpolating Variables Into a String

In Python version 3.6, a new string formatting mechanism was introduced. This feature is formally named the Formatted String Literal, but is more usually referred to by its nickname **f-string**.

The formatting capability provided by f-strings is extensive and won't be covered in full detail here. If you want to learn more, you can check out the Real Python article Python's F-String for String Interpolation and Formatting. There is also a tutorial on Formatted Output coming up later in this series that digs deeper into f-strings.

One simple feature of f-strings you can start using right away is variable interpolation. You can specify a variable name directly within an f-string literal, and Python will replace the name with the corresponding value.

For example, suppose you want to display the result of an arithmetic calculation. You can do this with a straightforward print() statement, separating numeric values and string literals by commas:

>>> n = 20

>>> m = 25

*>>> prod = n * m*

>>> print('The product of', n, 'and', m, 'is', prod)

The product of 20 and 25 is 500

But this is cumbersome. To accomplish the same thing using an f-string:

- Specify either a lowercase f or uppercase F directly before the opening quote of the string literal. This tells Python it is an f-string instead of a standard string.
- Specify any variables to be interpolated in curly braces ({}).

Recast using an f-string, the above example looks much cleaner:

>>> n = 20

>>> m = 25

>>> prod = n * m

>>> print(f'The product of {n} and {m} is {prod}')

The product of 20 and 25 is 500

Any of Python's three quoting mechanisms can be used to define an f-string:

>>> var = 'Bark'

>>> print(f'A dog says {var}!')

A dog says Bark!

>>> print(f"A dog says {var}!")

A dog says Bark!

>>> print(f'''A dog says {var}!''')

A dog says Bark!

1.4.8 Modifying Strings

In a nutshell, you can't. Strings are one of the data types Python considers immutable, meaning not able to be changed. In fact, all the data types you have seen so far are immutable. (Python does provide data types that are mutable, as you will soon see.)

A statement like this will cause an error:

>>> s = 'foobar'

>>> s[3] = 'x'

```
Traceback (most recent call last):
File "<pyshell#40>", line 1, in <module>
s[3] = 'x'
TypeError: 'str' object does not support item assignment
```

In truth, there really isn't much need to modify strings. You can usually easily accomplish what you want by generating a copy of the original string that has the desired change in place. There are very many ways to do this in Python. Here is one possibility:

```
>>> s = s[:3] + 'x' + s[4:]
>>> s
'fooxar'
```

There is also a built-in string method to accomplish this:

```
>>> s = 'foobar'
>>> s = s.replace('b', 'x')
>>> s
'fooxar'
```

2 Introduction to Collection

Data structures like lists, tuples, arrays, dictionaries, etc. are examples of data structures that are usually referred to as collections in Python.

A built-in collections module in Python offers extra data structures for data collections.

The six most popular data structures are included in the collection modules.

2.1 Defaultdict

A dictionary in Python is just like Defaultdict. The only distinction is that when you attempt to access a key that doesn't exist, it doesn't throw an exception or a key error.

Even though the 4th index in the following code wasn't initialized, the compiler still gives us a value of 0 when we try to access it.

EXAMPLE :

```
from collections import defaultdict
nums = defaultdict(int)
nums['one'] = 1
nums['two'] = 2
nums['three'] = 3
print(nums['four'])
```

OUTPUT:

0

2.2 Counter

Counter A built-in data structure known as a counter is used to track the frequency of each value in an array or list.

The following code counts how many times the value 2 appears in the provided list.

EXAMPLE:

```
from collections import Counter
list = [1,2,3,4,1,2,6,7,3,8,1,2,2]
answer=Counter()
answer = Counter(list)
print(answer[2])
```

OUTPUT :

4

2.3 Deque

Deque is the best form of a list for adding and removing items. It can add/remove items from either start or the end of the list.

The supplied list is being added to at the conclusion of the following code, and g is at the beginning of the same list.

EXAMPLE:

```
from collections import deque
#initialization
list = ["a","b","c"]
deq = deque(list)
print(deq)

#insertion
deq.append("z")
deq.appendleft("g")
print(deq)
#removal
deq.pop()
deq.popleft()
```

```
print(deq)
```

OUTPUT:

```
deque(['a', 'b', 'c'])
deque(['g', 'a', 'b', 'c', 'z'])
deque(['a', 'b', 'c'])
```

2.4 Namedtuple ()

A very significant issue in the world of computer science is resolved by the Namedtuple() function. Unlike namedtuple(), which returns names for each place in the tuple, regular tuples don't need to remember the index of each field of a tuple object.

In the code below, passing an attribute is sufficient to produce the desired output and does not require an index to print a student's name.

EXAMPLE

```
from collections import namedtuple
Student = namedtuple('Student', 'fname, lname, age')
s1 = Student('Peter', 'James', '13')
print(s1.fname)
```

OUTPUT

```
Peter
```

2.5 ChainMap

ChainMap creates a list of dictionaries by combining numerous dictionaries. With no limitations on the number of dictionaries, ChainMaps essentially condenses several dictionaries into a single unit.

The next application, ChainMap, will give you back two dictionaries.

EXAMPLE

```
import collections

dictionary1 = { 'a' : 1, 'b' : 2 }
dictionary2 = { 'c' : 3, 'b' : 4 }
chain_Map = collections.ChainMap(dictionary1, dictionary2)
print(chain_Map.maps)
```

OUTPUT

[{'a': 1, 'b': 2}, {'b': 4, 'c': 3}]

2.6 OrderedDict

OrderedDict A dictionary that keeps its order is called OrderedDict. For instance, the order is maintained if the keys are put in a particular order. The location will not change even if you later modify the key's value.

EXAMPLE

```
from collections import OrderedDict
order = OrderedDict()
order['a'] = 1
order['b'] = 2
order['c'] = 3
print(order)

#unordered dictionary
unordered=dict()
unordered['a'] = 1
unordered['b'] = 2
unordered['c'] = 3
print("Default dictionary", unordered)
```

OUTPUT

```
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
Default dictionary {'b': 2, 'a': 1, 'c': 3}
```

3 list manipulating

In python, there are several built-in data types to store collections of data. These include List, Tuples, Sets, Dictionary, Array, and Deques.

3.1 List:

A list is an ordered collection of elements, where each element can be of any data type. Lists are indexed, allow duplicate values, and are **mutable**, that is we can add, remove, or modify elements in the list. A list can be created using square brackets to enclose a comma-separated list of values, which can be numbers, strings, booleans, and other lists. An empty list my_list can be created by using my_list = [].

```
my_list = []
print("my_list is :", my_list)
```

```
print("type of my list is :", type(my_list))
```

OUTPUT

```
my_list is : []
type of my list is : <class 'list'>
```

List Index:

The elements in the list are indexed, the index of the first element will be zero, and the last element by an index of -1.

If I have a list of integers as my_list = [10, 20, 30, 40, 50], then 10 will have an index of 0, and 20 will have 1 and it goes like that. Element 50 can be retrieved by using index -1.

```
my_list = [10, 20, 30, 40, 50]
print("Element at index 0 is :", my_list[0])
print("Last element in list is :", my_list[-1])
```

OUTPUT

```
Element at index 0 is : 10
Last element in list is : 50
```

The index of any element can be obtained by using index().

```
print("Index of element 30 is :", my_list.index(30))
```

OUTPUT

```
Index of element 30 is : 2
```

The first two elements can be retrieved by using my_list[0:2] or my_list[:2]. Here the element will start from 0 indexes till index 2, Index 2 will not be included.

```
print("Elements from index 0 till index 2 is :", my_list[:2])
print("Elements from index 0 till index 2 is :", my_list[0:2])
```

OUTPUT

```
Elements from index 0 till index 2 is : [10, 20]
Elements from index 0 till index 2 is : [10, 20]
```

Add a New Element to the List:

There are a few ways to add an element at the end of the list.

1. **append():** we can add the element at the end of the list by using the function append().

```
my_list.append(60)
print("My new list is :", my_list)
```

OUTPUT

My new list is : [10, 20, 30, 40, 50, 60]

2. By using the addition of a list, if I want to add 70 to the existing list, I have to add 70 by using a square bracket. While adding an integer to a list, we have to make the integer as a list, else we will get a type error.

my_list += [70]

print("My new list is :", my_list)

OUTPUT

My new list is : [10, 20, 30, 40, 50, 70]

But if we add a string to the existing list by using the addition method, the string will be converted to a list of characters and then it will be added to the list, in case we are having space between characters, then the number space will be added as ' '. If I am trying to add 'my list' to[10, 20, 30, 40, 50, 60, 70] with 3 spaces in between 'my' and 'list' then the list formed will be [10, 20, 30, 40, 50, 60, 70, 'm', 'y', ' ', ' ', ' ', 'l', 'i', 's', 't']

print("My Original is :", my_list)

my_list += 'my list'

print("My new list with strings added is :", my_list)

OUTPUT

My Original is : [10, 20, 30, 40, 50, 60, 70]

My new list with strings added is : [10, 20, 30, 40, 50, 60, 70, 'm', 'y', ' ', ' ', ' ', 'l', 'i', 's', 't']

3. insert()

If we want to add an element at a specific index then we can use insert(index, element_to_add). For example, if I want to add an element 25 to my_list [10, 20, 30, 40, 50, 60, 70] at the 3rd position, that is at index 2, it can be done by using my_list.insert(2, 25).

print("My Original is :", my_list)

my_list.insert(2, 25)

print("My new list is after inserting 25 at index 2 is :", my_list)

OUTPUT

My Original is : [10, 20, 30, 40, 50, 60, 70]

My new list is after inserting 25 at index 2 is : [10, 20, 25, 30, 40, 50, 60, 70]

Remove an element from the List:

To remove an element from the list, if we know the index of the element to be removed, we use pop(), whereas if we know only the element which is to be removed we can use remove()

1. pop()

If we want to remove the last element of the list we can use pop(-1).

print("My Original is :", my_list)

my_list.pop(-1)

print("My list after removing last element", my_list)

OUTPUT

My Original is : [10, 20, 25, 30, 40, 50, 60, 70]

My list after removing last element [10, 20, 25, 30, 40, 50, 60]

2. remove()

If we want to remove 25 from my_list, it can be done as below.

print("My Original is :", my_list)

my_list.remove(25)

print("My list after removing element 25 is :", my_list)

OUTPUT

My Original is : [10, 20, 25, 30, 40, 50, 60]

My list after removing element 25 is : [10, 20, 30, 40, 50, 60]

Minimum and Maximum of the list of elements.

For a list of numbers, we can find the minimum by using min() and the maximum by using max().

print("My Original is :", my_list)

print("Greatest Element in my list is :", max(my_list))

print("Minimum Element in my list is :", min(my_list))

OUTPUT

My Original is : [10, 20, 30, 40, 50, 60]

Greatest Element in my list is : 60

Minimum Element in my list is : 10

Statistical Inference:

To find the mean, median, mode, and standard deviation we have to import the statistics library by using the command "import statistics".

import statistics

my_list += [20, 20]

print("My Original is :", my_list)

print("Mean of My List is :", statistics.mean(my_list))

print("Median of My List is :", statistics.median(my_list))

print("Mode of My List is :", statistics.mode(my_list))

print("Standard Deviation of My List is :", statistics.stdev(my_list))

OUTPUT

My Original is : [10, 20, 30, 40, 50, 60, 20, 20]

Mean of My List is : 31.25

Median of My List is : 25.0

Mode of My List is : 20

Standard Deviation of My List is : 17.268882005337975

Sorting A List:

To sort an unsorted list, my_list.sort() can be used, which will sort() the original list, whereas if we use sorted(my_list) will return a sorted list, but will not modify the original list. To get a sorted list from the latter method, we have to assign it to a new list.

sort in ascending order:

By simply using my_list.sort(), we can sort the list in ascending order, because the default value of the reverse parameter is False, or you can also use my_list.sort(reverse = False). For example, unsorted lists [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20], can be sorted as below.

```
my_list = [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20]
print("My Original is :", my_list)
my_list.sort()
print("My Sorted Listed is :", my_list)
```

OUTPUT

My Original is : [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20]

My Sorted Listed is : [1, 1, 1, 4, 4, 6, 20, 22, 39, 54, 78]

If we have a string, then it will sort using the alphabetic order.

```
fruit_list = ['apple', 'mangoes', 'banana', 'strawberry', 'pineapple']
print("My Original Fruit Listed is :", fruit_list)
fruit_list.sort()
print("My Sorted Fruit Listed is :", fruit_list)
```

OUTPUT

My Original Fruit Listed is : ['apple', 'mangoes', 'banana', 'strawberry', 'pineapple']

My Sorted Fruit Listed is : ['apple', 'banana', 'mangoes', 'pineapple', 'strawberry']

But when we use sorted(my_list), **the original list will not be modified,** we have assigned, it to a new_list.

```
my_list = [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20]
print("My Original is :", my_list)
sorted(my_list)
print("My list is not sorted using sorted() :", my_list)
```

my_new_list = sorted(my_list)

print("My List assigned to new list :", my_new_list)

OUTPUT

My Original is : [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20]

My list is not sorted using sorted() : [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20]

My List assinged to new list : [1, 1, 1, 4, 4, 6, 20, 22, 39, 54, 78]

Sort in descending order:

To sort in descending order we have to set the parameter reverse as True. For the above list, we can sort in descending order as below.

my_list = [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20]

print("My Original is :", my_list)

my_list.sort(reverse=True)

print("My reverse sorted list :", my_list)

fruit_list = ['apple', 'mangoes', 'banana', 'strawberry', 'pineapple']

print("My Original Fruit Listed is :", fruit_list)

fruit_list.sort(reverse=True)

print("My Reverse Sorted Fruit Listed is :", fruit_list)

OUTPUT

My Original is : [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20]

My reverse sorted list : [78, 54, 39, 22, 20, 6, 4, 4, 1, 1, 1]

My Original Fruit Listed is : ['apple', 'mangoes', 'banana', 'strawberry', 'pineapple']

My Reverse Sorted Fruit Listed is : ['strawberry', 'pineapple', 'mangoes', 'banana', 'apple']

For the sorted() function also we can sort in descending order by setting the parameter reverse as True.

print("My Original is :", my_list)

print("My list reverse sorted using sorted() :", sorted(my_list, reverse=True))

print("My Original Fruit Listed is :", fruit_list)

print("My fruit list reverse sorted using sorted() :", sorted(fruit_list, reverse=True))

OUTPUT

My Original is : [4, 6, 1, 4, 1, 22, 78, 54, 39, 1, 20]

My list reverse sorted using sorted() : [78, 54, 39, 22, 20, 6, 4, 4, 1, 1, 1]

My Original Fruit Listed is : ['apple', 'mangoes', 'banana', 'strawberry', 'pineapple']

My fruit list reverse sorted using sorted() : ['strawberry', 'pineapple', 'mangoes', 'banana', 'apple']

4 Collections Lists Tuples

Collections in python can be referred as a container which is used to store collection of data. There are some built-in collections such as list, tuple, set and dictionary.

LIST	TUPLE	DICTIONARY	SET
Allows duplicate members	Allows duplicate members	No duplicate members	No duplicate members
Changeble	Not changeable	Changeable, indexed	Cannot be changed, but can be added, non -indexed
Ordered	Ordered	Unordered	Unordered
Square bracket [ ]	Round brackets ( )	Curly brackets{ }	Curly brackets{ }

LIST

Lists are a used for preserving a sequence of data and then need not to be homogenous all the time, and are ordered and mutable i.e it can be changed after being created.

Functions for **List**-

1.	len()	how many elements a list have	len(my_list)
2.	append()	Adds item to the last of the list	my_list.append("ram")
3.	insert()	Adds item to a specified index	my_list.insert(1,"ram")
4	clear()	Its make the list empty	my_list.clear()
5.	del	Deletes the list or delete items from the specified index	del mylist del my_list[0]
6.	remove()	Removes the given item	my_list.remove("ram")
7.	pop()	Removes the item from the specified index Last if not specified	my_list.pop()
8.	copy()	To copy a list	ylist= my_list.copy()
9.	list()	To copy a list	ylist= list(my_list)
10.	extend()	To copy a list	list7.extends(list8)
11.	+	To copy a list	list5= list 1+list3
12 (2)	append	It can also be used for copying the list	For i in my_list: ylist= list.append(i)
13. (2)	List	Can also for creating a list	my_list= list(("a", "b", "c"))
14.	count()	How many values are there of the given item	x= points.count(6) x=names.count("ram")
15.	index()	Returns the index value of specified	X = points.index(5)
16.	sort()	In ascending order	names.sort()
17.	reverse()	Sorting in reverse order	names.reverse()
18.	Loop (for), (if)	Each element	For x in my_list: Print (x),, If "ram" in my_list print("fytf")

TUPLE

Tuples are the collections in python which are immutable i.e. cannot be changed and are ordered.

Functions for **Tuple** -

1.	Loop (for),(if)	Accessing	For x in the_tuple: print(x) If “ram” in the _tuple Print(“nriuj”)
2.	len	No. of items in given tuple	len(the_tuple)
3.	+	Join two tuple	tuple4= tuple2+tuple3
4.	tuple()	To make a tuple	ytuple=tuple((“a”,”b”,” c”))
5.	count()	Count the no of elements of given item	x= points.count(“ram”)
6.	index()	Returns the index number	x= points .index(“ram”)

To change value in tuple, first converted into list and then again in tuple

x =(“a”, “b”, “c”)

y = list(x)

y [2] = “h”

x = tuple (y)

5 Introduction to Tuples

In Python, a **tuple** is a built-in data type that allows you to create **immutable sequences** of values. The values or items in a tuple can be of any type. This makes tuples pretty useful in those situations where you need to store heterogeneous data, like that in a database record, for example.

Through this tutorial, you'll dive deep into Python tuples and get a solid understanding of their key features and use cases. This knowledge will allow you to write more efficient and reliable code by taking advantage of tuples.

In this tutorial, you'll learn how to:

- **Create** tuples in Python
- **Access** the items in an existing tuple
- **Unpack**, **return**, **copy**, and **concatenate** tuples
- **Reverse**, **sort**, and **traverse** existing tuples
- Explore other **features** and common **gotchas** of tuples

In addition, you'll explore some alternative tools that you can use to replace tuples and make your code more readable and explicit.

To get the most out of this tutorial, you should have a good understanding of a few Python concepts, including variables, functions, and for loops. Familiarity with other built-in data structures, especially lists, is also a plus.

5.1 Tuple Data Type

The built-in tuple data type is probably the most elementary sequence available in Python. Tuples are immutable and can store a fixed number of items. For example, you can use tuples to represent Cartesian coordinates (x, y), RGB colors (red, green, blue), records in a database table (name, age, job), and many other sequences of values.

In all these use cases, the number of elements in the underlying tuple is *fixed*, and the items are *unchangeable*. You may find several situations where these two characteristics are desirable. For example, consider the RGB color example:

>>> red = (255, 0, 0)

Once you've defined red, then you won't need to add or change any components. Why? If you change the value of one component, then you won't have a pure red color anymore, and your variable name will be misleading. If you add a new component, then your color won't be an RGB color. So, tuples are perfect for representing this type of object.

Note: Throughout this tutorial, you'll find the terms **items**, **elements**, and **values** used interchangeably to refer to the objects stored in a tuple.

Some of the most relevant characteristics of tuple objects include the following:

- **Ordered**: They contain elements that are sequentially arranged according to their specific insertion order.
- **Lightweight**: They consume relatively small amounts of memory compared to other sequences like lists.
- **Indexable through a zero-based index**: They allow you to access their elements by integer indices that start from zero.
- **Immutable**: They don't support in-place mutations or changes to their contained elements. They don't support growing or shrinking operations.
- **Heterogeneous**: They can store objects of different data types and domains, including mutable objects.
- **Nestable**: They can contain other tuples, so you can have tuples of tuples.
- **Iterable**: They support iteration, so you can traverse them using a loop or comprehension while you perform operations with each of their elements.
- **Sliceable**: They support slicing operations, meaning that you can extract a series of elements from a tuple.
- **Combinable**: They support concatenation operations, so you can combine two or more tuples using the concatenation operators, which creates a new tuple.
- **Hashable**: They can work as keys in dictionaries when all the tuple items are immutable.

Tuples are sequences of objects. They're commonly called **containers** or **collections** because a single tuple can contain or collect an arbitrary number of other objects.

Note: In Python, tuples support several operations that are common to other sequence types, such as lists, strings, and ranges. These operations are known as common sequence operations. Throughout this tutorial, you'll learn about several operations that fall into this category.

In Python, tuples are ordered, which means that they keep their elements in the original insertion order:

```
>>> record = ("John", 35, "Python Developer")

>>> record
('John', 35, 'Python Developer')
```

The items in this tuple are objects of different data types representing a record of data from a database table. If you access the tuple object, then you'll see that the data items keep the same original insertion order. This order remains unchanged during the tuple's lifetime.

You can access individual objects in a tuple by position, or index. These indices start from zero:

```
>>> record[0]
'John'
>>> record[1]
35
>>> record[2]
'Python Developer'
```

Positions are numbered from zero to the length of the tuple minus one. The element at index 0 is the first element in the tuple, the element at index 1 is the second, and so on.

5.2 Constructing Tuples in Python

A tuple is a sequence of comma-separated objects. To store objects in a tuple, you need to create the tuple object with all its content at one time. You'll have a couple of ways to create tuples in Python. For example, you can create tuples using one of the following alternatives:

- Tuple literals
- The tuple() constructor

In the following sections, you'll learn how to use the tools listed above to create new tuples in your code. You'll start off with tuple literals.

5.2.1 Creating Tuples Through Literals

Tuple literals are probably the most common way to create tuples in Python. These literals are fairly straightforward. They consist of a comma-separated series of objects.

Here's the general syntax of a tuple literal:

item_0, item_1, ..., item_n

This syntax creates a tuple of n items by listing the items in a comma-separated sequence. Note that you don't have to declare the items' type or the tuple's size beforehand. Python takes care of this for you.

In most situations, you'll create tuples as a series of comma-separated values surrounded by a pair of parentheses:

```
(item_0, item_1, ..., item_n)
```

The pair of parentheses in this construct isn't required. However, in most cases, the parentheses improve your code's readability. So, using the parentheses is a best practice that you'll see in many codebases out there. In contrast, the *commas* are required in the tuple literal syntax.

Here are a few examples of creating tuples through literals:

```
>>> jane = ("Jane Doe", 25, 1.75, "Canada")
>>> point = (2, 7)
>>> pen = (2, "Solid", True)

>>> days = (
... "Monday",
... "Tuesday",
... "Wednesday",
... "Thursday",
... "Friday",
... "Saturday",
... "Sunday",
... )
```

In the first three examples, you create tuples of heterogeneous objects that include strings, numbers, and Boolean values. Note that in these examples, each tuple represents a single object with different elements. So, the name of the underlying tuple is a singular noun.

In the final example, you create a tuple of homogeneous objects. All the items are strings representing the weekdays. The name of the tuple is a plural noun.

In the case of days, you should note that Python ignores any extra comma at the end of a tuple, as it happens after "Sunday". So, it's optional but common practice because it allows you to quickly add a new item if needed. It's also the default format that code formatters like Black apply to multiline tuples.

Note: In all of the above examples, the tuples have a fixed number of items. Those items are mostly constant in time, which means that you don't have to change or update them during your code's execution. This idea of a *fixed and unchangeable series of values* is the key to deciding when to use a tuple in your code.

Even though the parentheses aren't necessary to define most tuples, you do have to include them when creating an *empty* tuple:

```
>>> empty = ()
>>> empty
()

>>> type(empty)
<class 'tuple'>
```

Note that once you've created an empty tuple, you can't populate it with new data as you can do with lists. Remember that tuples are immutable. So, why would you need empty tuples?

For example, say that you have a function that builds and returns a tuple. In some situations, the function doesn't produce items for the resulting tuple. In this case, you can return the empty tuple to keep your function consistent regarding its return type.

You'll find a couple of other situations where using the parentheses is required. For example, you need it when you're interpolating values in a string using the % operator:

```
>>> "Hello, %s! You're %s years old." % ("Linda", 24)
'Hello, Linda! You're 24 years old.'

>>> "Hello, %s! You're %s years old." % "Linda", 24
Traceback (most recent call last):
...
TypeError: not enough arguments for format string
```

In the first example, you use a tuple wrapped in parentheses as the right-hand operand to the % operator. In this case, the interpolation works as expected. In the second example, you don't wrap the tuple in parentheses, and you get an error.

Another distinctive feature of tuple literals appears when you need to create a single-item tuple. Remember that the comma is the only required part of the syntax. So, how would you define a tuple with a single item? Here's the answer:

```
>>> one_word = "Hello",
>>> one_word
('Hello',)

>>> one_number = (42,)
>>> one_number
(42,)
```

To create a tuple with a single item, you need to place the item followed by a comma. In this example, you define two tuples using this pattern. Again, the parentheses aren't required. However, the trailing comma is required.

Single-item tuples are quite useful. For example, if you have a class that generates a large number of instances, then a recommended practice would be to use the .__slots__ special attribute in order to save memory. You'll typically use a tuple as the value of this attribute. If your class has only one instance attribute, then you'll define .__slots__ as a single-item tuple.

5.2.2 Using the tuple() Constructor

You can also use the tuple() class constructor to create tuple objects from an iterable, such as a list, set, dictionary, or string. If you call the constructor without arguments, then it'll build an empty tuple.

Here's the general syntax:

tuple([iterable])

To create a tuple, you need to call tuple() as you'd call any class constructor or function. Note that the square brackets around iterable mean that the argument is optional, so the brackets aren't part of the syntax.

Here are a few examples of how to use the tuple() constructor:

```
>>> tuple(["Jane Doe", 25, 1.75, "Canada"])
('Jane Doe', 25, 1.75, 'Canada')

>>> tuple("Pythonista")
('P', 'y', 't', 'h', 'o', 'n', 'i', 's', 't', 'a')

>>> tuple({
... "manufacturer": "Boeing",
... "model": "747",
... "passengers": 416,
... }.values())
('Boeing', '747', 416)

>>> tuple()
()
```

In these examples, you create different tuples using the tuple() constructor, which accepts any type of iterable object.

Note: The tuple constructor also accepts sets. However, remember that sets are unordered data structures. This characteristic will affect the final order of items in the resulting tuple.

Finally, note that calling tuple() without an argument returns a new empty tuple. This way of creating empty tuples is rare in practice. However, it can be more explicit and help you communicate your intent: *creating an empty tuple*. But in most cases, assigning an empty pair of parentheses to a variable is okay.

The tuple() constructor comes in handy when you need to create a tuple out of an iterator object. An iterator yields items on demand. So, you don't have access to all of its data at one time. The tuple() constructor will consume the iterator, build a tuple from its data, and return it back to you.

Here's an example of using the tuple() constructor to create a tuple out of a generator expression, which is a special kind of iterator:

*>>> tuple(x**2 for x in range(10))*

(0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

In this example, you use tuple() to build a tuple of square values. The argument to tuple() is a generator expression that yields square values on demand. The tuple constructor consumes the generator and builds the tuple containing all the data.

Note: It's important to note that to create a stand-alone generator expression, you do need an enclosing pair of parentheses. In the above example, the required parentheses are provided by the call to tuple().

You could've also done something like tuple((x**2 for x in range(10))), but this would be less readable and clean.

As a side note, you need to consider that potentially infinite iterators will hang your code if you feed them to the tuple() constructor.

5.3 Accessing Items in a Tuple: Indexing

You can extract the items of a tuple using their associated **indices**. What's an index? Each item in a tuple has an integer index that specifies its position in the tuple. Indices start at 0 and go up to the number of items in the tuple minus 1.

To access an item through its index, you can use the following syntax:

tuple_object[index]

This construct is known as an **indexing** operation. The [index] part is the **indexing operator**, which consists of a pair of square brackets enclosing the target index. You can read this construct as *from tuple_object give me the item at index.*

Here's how this syntax works in practice:

>>> jane = ("Jane Doe", 25, 1.75, "Canada")

>>> jane[0]

'Jane Doe'

>>> jane[1]

25

>>> jane[3]

'Canada'

Indexing a tuple with different indices gives you direct access to the associated values. If you use Big O notation for time complexity, then you can say that indexing is an *O(1)* operation. This means that tuples are quite good for those situations where you need to quickly access specific items from a series.

Here's a visual representation of how indices map to items in a tuple:

"Jane Doe"	**25**	**1.75**	**"Canada"**
0	1	2	3

In any Python tuple, the index of the first item is 0, the index of the second item is 1, and so on. The index of the last item is the number of items minus 1. In this example, the tuple has four items, so the last item's index is 4 - 1 = 3.

The number of items in a tuple defines its **length**. You can learn this number by using the built-in len() function:

>>> len(jane)

4

With a tuple as an argument, the len() function returns a value representing the number of items in the target tuple. This number is the tuple's length.

It's important to note that, if you use an index greater than or equal to the tuple's length, then you get an IndexError exception:

>>> jane[4]

Traceback (most recent call last):

...

IndexError: tuple index out of range

In this example, you get an IndexError as a result. Using out-of-range indices might be a common issue when you're starting to use tuples or other sequences in Python. So, keep in mind that indices are zero-based, so the last item in this example has an index of 3.

You can also use negative indices while indexing tuples. This feature is common to all Python sequences, such as lists and strings. Negative indices give you access to the tuple items in backward order:

>>> jane[-1]

'Canada'

>>> jane[-2]

1.75

A negative index specifies an element's position relative to the right end of the tuple and back to the beginning. Here's a representation of how negative indices work:

"Jane Doe"	**25**	**1.75**	**"Canada"**
-4	-3	-2	-1

You can access the last item in a tuple using the index -1. Similarly, the index -2 identifies the item next to the last, and so forth.

As you can see, negative indices don't start from 0. That's because 0 already points to the first item. This may be confusing when you're first learning about negative and positive indices. Don't worry, you'll get used to this behavior.

If you use negative indices, then -len(tuple_object) will be the first item in the tuple. If you use an index lower than this value, then you'll get an IndexError:

```
>>> jane[-5]
Traceback (most recent call last):
...
IndexError: tuple index out of range
```

Using an index lower than -len(tuple_object) produces an error because the target index is out of range.

As you already know, tuples can contain items of any type, including other sequences. When you have a tuple that contains other sequences, you can access the items in any nested sequence by chaining indexing operations.

To illustrate, say that you have the following tuple:

```
>>> employee = (
... "John",
... 35,
... "Python Developer",
... ("Django", "Flask", "FastAPI", "CSS", "HTML"),
... )
```

Your employee tuple has an embedded tuple containing a series of skills. How can you access individual skills? You can use the following indexing syntax:

```
tuple_of_sequences[index_0][index_1]...[index_n]
```

The numbers at the end of each index represent the different levels of nesting in the tuple. So, to access individual skills in the employee tuple, you first need to access the last item and then access the desired skill:

```
>>> employee[-1][0]
'Django'

>>> employee[-1][1]
'Flask'
```

You can access items in the nested sequence by applying multiple indexing operations in a row. This syntax is extensible to other nested sequences like lists and strings. It's even valid for dictionaries, in which case you'll have to use keys instead of indices.

5.4 Manipulating Tuples

As mentioned earlier, tuples are immutable in Python, which means that the elements of a tuple cannot be modified once they are created. If you try to reassign a value to an element of a tuple, you will get a TypeError. For example:

my_tuple = (1, "Hello", 3.14)

my_tuple[0] = 2 # This will raise a TypeError

However, there are a few ways to work around this limitation and achieve the desired effect of modifying a tuple.

One way is to convert the tuple to a list, make the desired changes to the list, and then convert it back to a tuple. For example:

my_tuple = (1, "Hello", 3.14)

my_list = list(my_tuple)

my_list[0] = 2

my_tuple = tuple(my_list)

Another way is to create a new tuple with the desired values, and reassign the variable to the new tuple. For example:

my_tuple = (1, "Hello", 3.14)

my_tuple = (2, my_tuple[1], my_tuple[2])

While it is not possible to directly modify the elements of a tuple in Python, it is possible to work around this limitation by converting the tuple to a list, making the desired changes, and then converting it back to a tuple, or by creating a new tuple with the desired values.

5.5 Tuple Concatenation and Repetition

Tuple concatenation is the process of combining two or more tuples into a single tuple. This can be done using the addition operator (+). For example:

tuple1 = (1, 2, 3)

tuple2 = (4, 5, 6)

new_tuple = tuple1 + tuple2

print(new_tuple)

 # Output:

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

Tuple repetition is the process of repeating the elements of a tuple a certain number of times. This can be done using the multiplication operator (*). For example:

my_tuple = (1, 2, 3)

*repeated_tuple = my_tuple * 3*

print(repeated_tuple)

Output:

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

It's important to note that concatenation and repetition create new tuples, the original tuples remain unchanged.

Tuple concatenation and repetition can be achieved in Python using the addition operator (+) and the multiplication operator (*) respectively. These operations allow you to combine multiple tuples or repeat the elements of a single tuple to create new tuples.

5.6 Tuple Deletion and Tuple Clear

In Python, as tuples are immutable, it is not possible to delete a single element from a tuple. However, you can delete the entire tuple using the del statement. For example:

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

Another way to clear the elements of a tuple is to create an empty tuple and assign it to the same variable. For example:

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

While it is not possible to delete a single element from a tuple in Python, it is possible to delete the entire tuple using the del statement or by re-assigning an empty tuple to the same variable.

5.7 Tuple Methods

Tuples have a few built-in methods that can be useful for working with the data stored in them.

The count() method returns the number of occurrences of a specific element in the tuple. For example:

```
my_tuple = (1, 2, 3, 2, 1)
print(my_tuple.count(2))
```

 # *Output:*

2

The index() method returns the index of the first occurrence of a specific element in the tuple. **For example:**

```
my_tuple = (1, 2, 3, 2, 1)
print(my_tuple.index(3))
```

 # *Output:*

2

The len() function returns the number of elements in the tuple. For example:

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

```
print(len(my_tuple))
# Output:
3
```

The min() and max() function returns the minimum and maximum element from the tuple respectively. For example:

```
my_tuple = (3,2,1)
print(min(my_tuple))
# Output:
1
print(max(my_tuple))
# Output:
3
```

In conclusion, tuples in Python have a few built-in methods such as count(), index(),len(),min() and max() that can be useful for working with the data stored in them. These methods allow you to perform common tasks such as counting occurrences of an element, finding the index of an element, and finding the length of the tuple and the min or max element in the tuple.

5.8 Tuple Unpacking

Tuple unpacking is a feature that allows you to assign the elements of a tuple to multiple variables in a single line of code. This can make your code more concise and readable.

For example, if you have a tuple with three elements and you want to assign each element to a separate variable, you can use tuple unpacking like this:

```
my_tuple = (1, "Hello", 3.14)
a, b, c = my_tuple
print(a)
# Output:
1
print(b)
# Output:
"Hello"
print(c)
# Output:
3.14
```

You can also use tuple unpacking to swap the values of two variables in a single line of code:

```
a = 1
b = 2
```

```
a, b = b, a
print(a)
# Output:
2
print(b)
# Output:
1
```

Additionally, tuple unpacking can also be used when iterating over the items of an iterable. For example:

```
my_list = [(1, "a"), (2, "b"), (3, "c")]
for a, b in my_list:
print(a, b)
```

This will print

1 a

2 b

3 c

Tuple unpacking is a feature in Python that allows you to assign the elements of a tuple to multiple variables in a single line of code, making your code more concise and readable. It can also be used to swap values of variables, and when iterating over items of an iterable.

5.9 Tuple Comprehension

In Python, a tuple comprehension is a concise way to create a new tuple by applying a certain operation to each element of an existing iterable, such as a list or another tuple. It is similar to a list comprehension, but it creates a tuple instead of a list.

The syntax for a tuple comprehension is similar to a list comprehension, but it uses parentheses () instead of square brackets []. For example, to create a new tuple containing the squares of the numbers in an existing list, you can use the following tuple comprehension:

```
numbers = [1, 2, 3, 4, 5]
squares = (x**2 for x in numbers)
print(squares)
# Output:
(1, 4, 9, 16, 25)
```

You can also use an if statement to filter out certain elements from the original iterable:

```
numbers = [1, 2, 3, 4, 5]
even_squares = (x**2 for x in numbers if x % 2 == 0)
```

print(even_squares)

 # Output:

(4, 16)

You can also use a nested comprehension to iterate over multiple iterables and create a tuple of tuples:

list1 = [1, 2, 3]

list2 = [4, 5, 6]

tuples = ((x, y) for x in list1 for y in list2)

print(tuples)

 # Output:

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