Python Data structures and Files

Python has a few Python Specific Data Structures (List, Tuple, Dictionary) and non- Python Specific Data Structures (String, sets). All these data structures are quite sophisticated and help us to store a collection of values rather than a single one. For this course, we will discuss String, List, Tuple, and Dictionaries. Even though file reading is not part of the data structures, the ability to store and fetch previously-stored information is very essential for any language. SO we have covered it at the end of this chapter.

Strings

String is a sequence of ordered characters (alphabets-lower case, upper case, numeric values, special symbols) represented with quotation marks (single('), double(''), triple(' ' ', '' '' '')) at the beginning and end. The character order is always from left to right. For example: using single quotes ('CSE110'), double quotes ("CSE110") or triple quotes ('''CSE110''' or """CSE110"""), "Hello CSE110 students".

Single line String and Multi-line String

Strings can be made of a single line as well as multiple lines. For representing single lines, single(') or double('') quotation marks are used. Whereas for representing multiple lines, triple quotation marks (""") or three single quotation marks(''') **must** be used. The multiple line strings are usually called "Doc-Strings". Details about this will be discussed in the function chapter.

Example:	

Single line	`print("Loving CSE110 Course")`
	
	`Output: Loving CSE110 Course`

	`print('Loving CSE110 Course')`
	
	`Output: Loving CSE110 Course`

	`print('She said, "I love coding"')`
	
	`Output: She said, "I love coding"`

Multiple lines	`print("""We Love Coding in Python.`
	`But after mastering Python,`
	`should we call ourselves, "Snake charmers"`
	`or "Python programmers"?`
	`""")`
	
	`Output:`
	`We Love Coding in Python.`
	`But after mastering Python,`
	`should we call ourselves, "Snake charmers"`
	`or "Python programmers"?`

String printing special cases

1) In the last single line example, we had to print a pair of quotation marks as a part of the output. Since Python has 3 options (single('), double(''), triple(''', """) quotation) for representing String, we can easily print a pair quotation, using the alternative one. For example:

Code	Output
`print("I love 'Chocolates'")`	`I love 'Chocolates'`
`print('Craving for a "Dominos pizzas"')`	`Craving for a "Dominos pizzas"`

2) Assume, you have to print **She asked, "Aren't you late for work?".** Here you have to print a pair of quotation marks as well as one single quotation mark, so the previous way of string printing will not work here. This problem can be solved by using triple quotes(''', """), representing special characters with a preceding backslash which is called an **escape character**. Using an escape character, or a preceding backslash tells the interpreter to consider the character following the backslash as a printable character. For example, putting **\"** will print ", putting **\'** will print ', and putting **** will print \.

To print our example, we can use multiple variations of escape characters. We can either put the entire string (along with the double quotes) inside a single quote and then use the escape character to print the single quote mark in the middle. Or in the opposite manner, we can put the entire string in double-quotes, and print the double quote mark in the string using escape characters. For example:

Code	Output
`print('''She asked, "Aren\'t you late for work?"''')`	`She asked, "Aren't you late for work?"`
`print('She asked, "Aren\'t you late for work?"')`	`She asked, "Aren't you late for work?"`
`print("She asked, \"Aren't you late for work?\"")`	`She asked, "Aren't you late for work?"`

Note: Details about escape characters and escape sequences are provided in the link below.
Link: https://docs.python.org/2.0/ref/strings.htmlEmpty String

Empty String

Strings represented with only two single(' ') or two double("") quotes with no characters in between are called **Empty String.** For example:

Code	Output
`print("")` `print('')`	Shows nothing in the output, because empty strings had nothing in between their quotation marks.

String indexing

For accessing individual characters within a string, python uses square brackets to enclose the index value or the position of that particular character
Indexing can be done in two ways:

1. Positive indexing:
   It is used to access the characters from the left side of a string. Positive indexing always starts from 0 (left-most character of the string) and ends at the last character of the string.

2. Negative indexing:
   It is used to access the characters from the right side of a string. Negative indexing always starts from -1 (rightmost character of the string) and ends at the first character of the string.

Positive indexing	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
	L	**o**	**v**	**i**	**n**	**g**		**C**	**S**	**E**	**1**	**1**	**0**
Negative indexing	**-13**	**-12**	**-11**	**-10**	**-9**	**-8**	**-7**	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**

Here, in the example above indexing of “Loving CSE110” is shown. The positive indexing starts at 0 with ‘L’ and ends at ‘0’ with 12. The negative indexing starts at -1 with ‘0’ and ends at ‘L’ with -13. If anything out of the valid range is tried to be accessed, the program will yield an IndexError. Additionally, if any other data type except an integer is entered as an index, TypeError will rise.

The basic String indexing structure
`string_name[ Index_value]`

For example:

Code	Output
`text = "Loving CSE110"` `print(text[4])`	`n`
`print(text[-4])`	`E`
`print(text[13])`	`IndexError`
`print(text[-15])`	`IndexError`
`print(text[1.5])`	`TypeError`

String slicing

Slicing is used for getting a substring of a particular string. In more general terms, it is used for accessing a particular range of characters. Colon(:) is used as a slicing operator.

Basic structure of slicing
`string_name[ beginning : end : step_size ]`
• Here, beginning: the index where slicing starts (inclusive). If not provided, by default starts from index 0. • end: the index where slicing stops(Not inclusive). If not provided, by default includes the rest of the string after "beginning". • Step: increment of the index value. If not provided, by default the value is 1.

Example:

Code	Output
`text = "Loving CSE110"` `print(text[2:5])`	`vin` `prints from the 3nd character(index 2) to the 6th (index 5) character of the string`
`print(text[4:-3])`	`ng CSE` `prints from the 5th character (index 4) to the 3rd last character (index 10) of the string`

`print(text[2:12:3])`	`vgS1` prints from the 2nd character (index 2) to the 13th character (index 12) of the string, with a step size of 3.
`print(text[2:])`	`ving CSE110` prints from the 3rd character(index 2) to the very last character of the string
`print(text[:-3])`	`Loving CSE` prints from the first character(index 0) to the 3rd last character (index 10) of the string
`print(text[-9:-2])`	`ng CSE1` prints from the 9th last character of the string (index 4) to the 2nd last character (index 11) of the string

String Operators

Few of the basic operators discussed in the first chapter work quite differently while working with strings. In the table below, you will get an idea about their adapted working procedure with strings.

Operator	Meaning & Description	Example
+	Concatenation: Values are added from both side of the '+' operator	print("abc"+"de") Output: abcde
*	Repetition: a new string is created with the specified number of copies of the input string	print("abc"*4) Output: abcabcabcabc
string_name[Index_value]	String Indexing: as discussed above.	text = "Hello" print(text[1]) Output: e
string_name[beginning : end : step_size]	String slicing: as discussed above.	text = "Hello Students" print(text[3:9]) Output: lo Stu
in	Membership: Returns True if the first value is in the second. Otherwise, returns False.	"ph" in "elephant" Output: True

not in	Membership: Returns True if the first value is not in the second. Otherwise, returns False.	"ph" not in "elephant" Output: False
%	Format: used for formatting strings. Details will be discussed later.	Examples will be discussed later.

String Immutability

Immutable means once it has been created its value cannot be changed. So, each time we have to modify the values, we need to make a copy of the original one and make changes to the duplicate one. For example, the String's built-in methods. But it has an advantage that its elements can be accessed very fast. For example:

Code	Output
text = "Python" text[3] = "K"	TypeError: 'str' object does not support item assignment Explanation: here, we were trying to change the value of the 3rd index, 'h' to "K". Since python does not allow changes to immutable objects, we got an error in the output.

Here, from the example, we can see that Strings do not allow any kind of alteration to their values as Strings are immutable.

String methods

Python has a good number of built-in methods or functions to utilize Strings. Because of string immutable property, these methods do not change the original String rather returns a new String copy every time. Among these methods, the most frequently used ones have been mentioned in the table below.

Method name	Description	Example
len(string)	Returns the length of a string	print(len('Hello')) Output: 5
lower()	Returns a copy string with all lower case letters	text = 'Hello World' temp = text.lower() print(text) print(temp)

		Output: Hello World hello world
upper()	Returns a copy string with all upper case letters	text = 'Hello World' temp = text.upper() print(text) print(temp) Output: Hello World HELLO WORLD
strip()	Returns a copy string with all whitespace remove before and after letters	text = ' BracU CSE110 ' temp = text.strip() print(text) print(temp) Output: BracU CSE110 BracU CSE110
lstrip()	Returns a copy string with all whitespace remove before and after letters	text = ' BracU CSE110 ' temp = text.lstrip() print(text) print(temp) Output: BracU CSE110 BracU CSE110

rstrip()		text = ' BracU CSE110 '
		temp = text.rstrip()
		print(text)
		print(temp)
		
		Output:
		  BracU CSE110
		  BracU CSE110

count(substring)	Returns total occurrence of a substring	text = 'Bangladesh' temp = text.count('a') print(text) print(temp)

		Output: Bangladesh 2
startswith(substring)	Returns True if the String starts with given substring	text = 'Hello' temp = text.startswith('He') print(text) print(temp) Output: Hello True
endswith(substring)	Returns True if the String ends with given substring	text = 'Hello' temp = text.startswith('hi') print(text) print(temp) Output: Hello False
find(substring)	Returns the index of first occurrence of substring	text = 'Bangladesh' temp = text.find('a') print(text) print(temp) Output: Bangladesh 1
replace(oldstring, newstring)	Replace every instance of oldstring with newstring	text = 'Hello' temp = text.replace('l', 'nt') print(text) print(temp) Output: Hello Hentnto

Note: Details about String methods will be found in the link below.
Link: https://docs.python.org/2/library/stdtypes.html#string-methods

ASCII

ASCII stands for American Standard Code for Information Interchange. In order to store and manipulate data, it needs to be in binary values. Here, in ASCII 128 English characters and different symbols are represented with a 7-bit binary number, with a decimal value ranging from 0 to 127. For example, 65 is the ASCII value of the letter 'A', and 97 is the ASCII value of the letter 'a'.

Two built-in Python functions ord() and chr() are used to make the conversions between ASCII values and characters. The ord() converts characters to ASCII values and chr() converts ASCII values to characters.

Example:	**Output**
`chr(98)`	`'b'`
`chr(66)`	`'B'`
`ord('a')`	`97`
`ord('%')`	`37`

Note: The ASCII chart will be found in the link below.
Link: http://www.asciitable.com/

Lists

Data structures are containers that organize and group data types together in different ways. A list in python is one of the most common and basic data structures that are written in square brackets "[]" with elements stored inside separated by commas. A list is also ordered and mutable. For example:

Code	Output

`list = ["Hi!", "Welcome", "to", "CSE", "110"]`
 	
`print(list)`
	`['Hi!', 'Welcome', 'to', 'CSE', '110']`

Mutability

The word "mutability" in python means the ability for certain types of data to be changed without recreating it entirely. It makes the program run more efficiently and quickly. For example:

Code	Output

`beatles = ["John", "Paul", "Alonzo", "Ringo"]`
 	
`#Before Modification`
`print(beatles)`
	`['John', 'Paul', 'Alonzo', 'Ringo']`

`beatles[2] = "George"`
 	
`#After Modification`
`print(beatles)`
	`['John', 'Paul', 'George', 'Ringo']`

Necessity of Lists

Lists don't need to be always homogeneous. They maintain the order of sequences whose values are not fixed. We can easily access and modify values inside a list. For example, adding value or removing is possible in a list. On the other hand, tuples are immutable and cannot be changed (you will get an explanation in the next section named "tuple"). Again, in python lists and strings are quite similar. We can use the "for" loop to iterate over lists, "+" plus operator in order to concatenate and use in a keyword to check if the sequence contains a value.

To access the Items of Lists:

1. Print the items in a list, we use index values:

Code	Output
`example_list = ["Kitkat", "Oreo", "Hersheys"]` `print(example_list[2])`	`Hersheys`

2. In a list, negative indexing means beginning from the end. Example: -1 refers to the last item, -2 refers to the second-last item, etc..

Code	Output
example_list = ["Kitkat", "Oreo","Hersheys"] print(example_list[-1]) print(example_list[-2])	Hersheys Oreo

3. While specifying a range in a list, the return value will give a new list with the specified items:

Code	Output
example_list = ["Hersheys","Kitkat", "Oreo", "MIMI", "Cadbury", "Monchuri-Milk Candy"] print(example_list[2:5])	['Oreo', 'MIMI', 'Cadbury']

4. While specifying a range of negative indexes in a list, we can start the search from the end of a list. Given example returns the items from index -5 (included) to index -1 (excluded).

Code	Output
example_list = ["Hersheys","Kitkat", "Oreo", "MIMI", "Cadbury", "Monchuri-Milk Candy"] print(example_list[-5:-1])	['Kitkat', 'Oreo', 'MIMI', 'Cadbury']

5. By leaving out the start value, the range will start from the first item of the list. Again, by leaving out the end value, the range will go on to the end of the list.

Code	Output
#Example-1)leaving out the start value: example_list = ["Hersheys","Kitkat", "Oreo", "MIMI", "Cadbury", "Monchuri-Milk Candy"] print(example_list[:3])	['Hersheys', 'Kitkat', 'Oreo']
#Example-2)leaving out the end value: example_list = ["Hersheys","Kitkat", "Oreo", "MIMI", "Cadbury", "Monchuri-Milk Candy"] print(example_list[4:])	['Cadbury', 'Monchuri-Milk Candy']

6. Since lists are mutable, we can change the items inside it.

Code	Output
example_list = ["Hersheys","Kitkat", "Oreo", "MIMI", "Cadbury", "Monchuri-Milk Candy"] print(example_list[2:5])	['Oreo', 'MIMI', 'Cadbury']

Basic Operations of List:

Example:	Output
#1) To make a copy of items of the list into a new list, using [:] operator: example_list = ["Hersheys","Kitkat", "Oreo", "MIMI", "Cadbury", "Monchuri-Milk Candy"] duplicate_list = example_list [:] print(duplicate_list)	['Hersheys', 'Kitkat', 'Oreo', 'MIMI', 'Cadbury', 'Monchuri-Milk Candy']
#2) The copied list remains unchanged when the item of original list gets modified: original_list = ["Hersheys","Kitkat", "Oreo", "MIMI", "Cadbury", "Monchuri-Milk Candy"] duplicate_list = original_list [:] original_list[-1] = "Pran Mango Bar" print(original_list) print (duplicate_list)	['Hersheys', 'Kitkat', 'Oreo', 'MIMI', 'Cadbury', **'Pran Mango Bar'**] ['Hersheys', 'Kitkat', 'Oreo', 'MIMI', 'Cadbury', **'Monchuri-Milk Candy'**]
#3) Looping through list using for loop: example_list = ["Hersheys","Kitkat", "Oreo", "MIMI"] for i in example_list: print(i)	Hersheys Kitkat Oreo MIMI
#4) To check if an item exists in the tuple:	Yes,MIMI is in the list

`example_list = ["Hersheys","Kitkat", "Oreo", "MIMI"]`
 	
`if "MIMI" in example_list:`
`  print("Yes,MIMI is in the list")`	

#5) To know the number of items in a list:
 	
`example_list = ["Hersheys","Kitkat", "Oreo", "MIMI"]`
 	
`print(len(example_list))`
	`4`

#6) List membership Test:
 	
`example_list = ["Hersheys","Kitkat", "Oreo", "MIMI"]`
 	
`print('Hersheys' in example_list)`
 	
`print('Cadbury' in example_list)`
	`True`
	
	`False`

Methods Associated with Lists:

Methods Associated with Lists	
L.append(e)	adds the object e to the end of L.
L.count(e)	returns the number of times that e occurs in L.
L.insert(i,e)	inserts the object e into L at index i.
L.extend(L1)	adds the items in list L1 to the end of L.
L.remove(e)	deletes the first occurrence of e from L.
L.index(e)	returns the index of the first occurrence of e in L.
L.pop(i)	removes and returns the item at index i in L. If i is omitted, it defaults to -1, to remove and return the last element of L.
L.sort()	sorts the elements of L in ascending order.
L.reverse()	reverses the order of the elements in L.

Built-in Methods of Lists with Examples:

Method	Code	Output

append()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.append('Mechanical')`
	`print(subjects)`
		`['CSE', 'EEE', 'Civil', 'Mechanical']`

count()	`subjects = ['CSE', 'EEE', 'Civil', 'CSE']`
	`x = subjects.count('CSE')`
	`print(x)`
		`2`

insert()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.insert(1, 'Mechanical')`
	`print(subjects)`
		`['CSE', 'Mechanical', 'EEE', 'Civil']`

extend()	`subjects = ['CSE', 'EEE', 'Civil']`
	`cars = ['Ford', 'BMW', 'Volvo']`
	`subjects.extend(cars)`
	`print(subjects)`
		`['CSE', 'EEE', 'Civil', 'Ford', 'BMW', 'Volvo']`

remove()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.remove('Civil')`
	`print(subjects)`
		`['CSE', 'EEE']`

index()	`subjects = ['CSE', 'EEE', 'Civil']`
	`x = subjects.index('Civil')`
	`print(x)`
		`2`

pop()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.pop(1)`
	`print(subjects)`
		`['CSE', 'Civil']`

sort()	`cars = ['Ford', 'BMW', 'Volvo']`
	`cars.sort()`
	`print(cars)`
		`['BMW', 'Ford', 'Volvo']`

reverse()	`subjects = ['CSE', 'EEE', 'Civil']`
	`subjects.reverse()`
	`print(subjects)`
		`['Civil', 'EEE', 'CSE']`

Slicing

We can understand slicing by visualizing the index to be in between the elements as shown below. How the slicing works have been explained in detail in the String chapter. The slicing mechanism works identically in all the data structures.

String	F	A	N	T	A	S	T	I	C
Positive indexing	0	1	2	3	4	5	6	7	8
Negative indexing	-9	-8	-7	-6	-5	-4	-3	-2	-1

Slicing elements of a list in python

If we want to access a range, we need two indices that will slice that portion from the list. For example:

Example:	**Output**

`#Declare a List named 'my_list'`
 	
`my_list = ['F','A','N','T','A','S','T','I','C']`	

`#1) elements 3rd to 5th`
 	
`print(my_list[2:5])`
	`['N', 'T', 'A']`

`#2) elements beginning to 4th`
 	
`print(my_list[:-5])`
	`['F', 'A', 'N', 'T']`

`#3) elements 6th to end`
 	
`print(my_list[5:])`
	`['S', 'T', 'I', 'C']`

`#4) elements beginning to end`
 	
`print(my_list[:])`
	`['F', 'A', 'N', 'T', 'A', 'S', 'T', 'I', 'C']`

Style Guide for Python Code

For every programming language, there are few coding conventions followed by the coding community of that language. All those conventions or rules are stored in a collected document manner for the convenience of the coders, and it is called the “Style Guide” of that particular programming language. The provided link gives the style guidance for Python code comprising the standard library in the main Python distribution.

Python style guide link: https://www.python.org/dev/peps/pep-0008/