# Session 2 - Strings and list objects

## Strings

Strings are used to record the text information such as name. In Python, Strings act as “Sequence” which means Python tracks every element in the String as a sequence. This is one of the important features of the Python language.

For example, Python understands the string "hello' to be a sequence of letters in a specific order which means the indexing technique to grab particular letters (like first letter or the last letter).

## Creating a String

In Python, either single quote (’) or double quotes (“) must be used while creating a string.

```
For example:
```

In [1]:
```
# Single word
'hello'
```

Out[1]:
```
'hello'
```

In [2]:
```
# Entire phrase
'This is also a string'
```

Out[2]:
```
'This is also a string'
```

In [3]:
```
# We can also use double quote
"String built with double quotes"
```

Out[3]:
```
'String built with double quotes'
```

In [4]:
```
# Be careful with quotes!
' I'm using single quotes, but will create an error'
```

```
  File "<ipython-input-4-6565b0b7b5e3>", line 2
    ' I'm using single quotes, but will create an error'
        ^
SyntaxError: invalid syntax
```

The above code results in an error as the text “I’m” stops the string. Here, a combination of single quotes and double quotes can be used to get the complete statement.

In [5]:
```
"Now I'm ready to use the single quotes inside a string!"
```

Out[5]:
```
"Now I'm ready to use the single quotes inside a string!"
```

Now let's learn about printing strings!

## Printing a String

We can automatically display the output strings using Jupyter notebook with just a string in a cell. But,the correct way to display strings in your output is by using a print function.

In [6]:
```
# We can simply declare a string
'Hello World'
```

Out[6]:
```
'Hello World'
```

In [7]:
```
# note that we can't output multiple strings this way
'Hello World 1'
'Hello World 2'
```

Out[7]:
```
'Hello World 2'
```

In Python 2, the output of the below code snippet is displayed using "print" statement as shown in the below syntax but the same syntax will throw error in Python 3.

In [8]:
```
print 'Hello World 1'
print 'Hello World 2'
print 'Use \n to print a new line'
print '\n'
print 'See what I mean?'
```

```
  File "<ipython-input-8-30861e38aa5b>", line 1
    print 'Hello World 1'
                        ^
SyntaxError: Missing parentheses in call to 'print'
```

## Python 3 Alert!

Note that, In Python 3, print is a function and not a statement. So you would print statements like this: print('Hello World')

If you want to use this functionality in Python2, you can import form the **future** module.

**Caution: After importing this; you won't be able to choose the print statement method anymore. So pick the right one whichever you prefer depending on your Python installation and continue on with it.**

In [9]:
```
# To use print function from Python 3 in Python 2
from __future__ import print_function

print('Hello World')
```

```
Hello World
```

## String Basics

In Strings, the length of the string can be found out by using a function called len().

In [10]:
```
len('Hello World')
```

```
Out[10]: 11
```

## String Indexing

We know strings are a sequence, which means Python can use indexes to call all the sequence parts. Let's learn how String Indexing works. • We use brackets [] after an object to call its index. • We should also note that indexing starts at 0 for Python. Now, Let's create a new object called s and the walk through a few examples of indexing.

```
In [11]: # Assign s as a string
         s = 'Hello World'
```

```
In [12]: #Check
         s

Out[12]: 'Hello World'
```

```
In [13]: # Print the object
         print(s)

         Hello World
```

Let's start indexing!

```
In [14]: # Show first element (in this case a letter)
         s[0]

Out[14]: 'H'
```

```
In [15]: s[1]

Out[15]: 'e'
```

```
In [16]: s[2]

Out[16]: 'l'
```

We can use a : to perform *slicing* which grabs everything up to a designated point. For example:

```
In [17]: # Grab everything past the first term all the way to the length of s which is len(s)
         s[1:]

Out[17]: 'ello World'
```

```
In [18]: # Note that there is no change to the original s
         s

Out[18]: 'Hello World'
```

```
In [19]: # Grab everything UP TO the 3rd index
```

```
s[:3]
```

Out[19]:
```
'Hel'
```

Note the above slicing. Here we're telling Python to grab everything from 0 up to 3. It doesn't include the 3rd index. You'll notice this a lot in Python, where statements and are usually in the context of "up to, but not including".

In [20]:
```
#Everything
s[:]
```

Out[20]:
```
'Hello World'
```

We can also use negative indexing to go backwards.

In [21]:
```
# Last letter (one index behind 0 so it loops back around)
s[-1]
```

Out[21]:
```
'd'
```

In [22]:
```
# Grab everything but the last letter
s[:-1]
```

Out[22]:
```
'Hello Worl'
```

Index and slice notation is used to grab elements of a sequenec by a specified step size (where in 1 is the default size). For instance we can use two colons in a row and then a number specifying the frequency to grab elements. For example:

In [23]:
```
# Grab everything, but go in steps size of 1
s[::1]
```

Out[23]:
```
'Hello World'
```

In [24]:
```
# Grab everything, but go in step sizes of 2
s[::2]
```

Out[24]:
```
'HloWrd'
```

In [25]:
```
# We can use this to print a string backwards
s[::-1]
```

Out[25]:
```
'dlroW olleH'
```

## String Properties

Immutability is one the finest string property whichh is created once and the elements within it cannot be changed or replaced. For example:

In [26]:
```
s
```

```
Out[26]: 'Hello World'
```

```
In [27]: # Let's try to change the first letter to 'x'
         s[0] = 'x'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-3a9c668aa5ab> in <module>()
      1 # Let's try to change the first letter to 'x'
----> 2 s[0] = 'x'

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

Notice how the error tells us directly what we can't do, change the item assignment!

Something we can do is concatenate strings!

```
In [28]: s

Out[28]: 'Hello World'
```

```
In [29]: # Concatenate strings!
         s + ' concatenate me!'

Out[29]: 'Hello World concatenate me!'
```

```
In [30]: # We can reassign s completely though!
         s = s + ' concatenate me!'
```

```
In [31]: print(s)

Hello World concatenate me!
```

```
In [32]: s

Out[32]: 'Hello World concatenate me!'
```

We can use the multiplication symbol to create repetition!

```
In [33]: letter = 'z'
```

```
In [34]: letter*10

Out[34]: 'zzzzzzzzzz'
```

## Basic Built-in String methods

In Python, Objects have built-in methods which means these methods are functions present inside the object (we will learn about these in much more depth later) that can perform actions or commands on the object itself.

Methods can be called with a period followed by the method name. Methods are in the form:

object.method(parameters)

Where parameters are extra arguments which are passed into the method. Right now, it is not necessary to make 100% sense but going forward we will create our own objects and functions.

Here are some examples of built-in methods in strings:

In [35]:
```
s
```
Out[35]:
```
'Hello World concatenate me!'
```

In [36]:
```
# Upper Case a string
s.upper()
```
Out[36]:
```
'HELLO WORLD CONCATENATE ME!'
```

In [37]:
```
# Lower case
s.lower()
```
Out[37]:
```
'hello world concatenate me!'
```

In [38]:
```
# Split a string by blank space (this is the default)
s.split()
```
Out[38]:
```
['Hello', 'World', 'concatenate', 'me!']
```

In [39]:
```
# Split by a specific element (doesn't include the element that was split on)
s.split('W')
```
Out[39]:
```
['Hello ', 'orld concatenate me!']
```

There are many more methods than the ones covered here. To know more about the String functions, Visit the advanced String section.

## Print Formatting

Print Formatting ".format()" method is used to add formatted objects to the printed string statements.

Let's see an example to clearly understand the concept.

In [40]:
```
'Insert another string with curly brackets: {}'.format('The inserted string')
```
Out[40]:
```
'Insert another string with curly brackets: The inserted string'
```

## Location and Counting

In [41]:
```
s.count('o')
```

Out[41]:
```
3
```

In [42]:
```
s.find('o')
```

Out[42]:
```
4
```

## Formatting

The center() method allows you to place your string 'centered' between a provided string with a certain length.

In [ ]:
```

```

expandtabs() will expand tab notations \t into spaces. Let's see an example to understand the concept.

In [ ]:
```

```

## is check methods

These various methods below check it the string is some case. Lets explore them:

In [45]:
```
s = 'hello'
```

isalnum() will return "True" if all characters in S are alphanumeric.

In [46]:
```
s.isalnum()
```

Out[46]:
```
True
```

isalpha() wil return "True" if all characters in S are alphabetic.

In [47]:
```
s.isalpha()
```

Out[47]:
```
True
```

islower() will return "True" if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

In [48]:
```
s.islower()
```

Out[48]:
```
True
```

isspace() will return "True" if all characters in S are whitespace.

In [49]:
```
s.isspace()
```

Out[49]:
```
False
```

istitle() will return "True" if S is a title cased string and there is at least one character in S, i.e. uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.

```
In [50]: s.istitle()
```

```
Out[50]: False
```

isupper() will return "True" if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

```
In [51]: s.isupper()
```

```
Out[51]: False
```

Another method is endswith() which is essentially same as a boolean check on s[-1]

```
In [52]: s.endswith('o')
```

```
Out[52]: True
```

## Built-in Reg. Expressions

In Strings, there are some built-in methods which is similar to regular expression operations. • Split() function is used to split the string at a certain element and return a list of the result. • Partition is used to return a tuple that includes the separator (the first occurrence), the first half and the end half.

```
In [53]: s.split('e')
```

```
Out[53]: ['h', 'llo']
```

```
In [54]: s.partition('e')
```

```
Out[54]: ('h', 'e', 'llo')
```

```
In [55]: s
```

```
Out[55]: 'hello'
```

# Lists

Earlier, while discussing introduction to strings we have introduced the concept of a *sequence* in Python. In Python, Lists can be considered as the most general version of a "sequence". Unlike strings, they are mutable which means the elements inside a list can be changed!

Lists are constructed with brackets [] and commas separating every element in the list.

Let's go ahead and see how we can construct lists!

```
In [1]:
# Assign a list to an variable named my_list
my_list = [1,2,3]
```

We just created a list of integers, but lists can actually hold different object types. For example:

```
In [2]:
my_list = ['A string',23,100.232,'o']
```

Just like strings, the len() function will tell you how many items are in the sequence of the list.

```
In [3]:
len(my_list)
```

```
Out[3]: 4
```

## Indexing and Slicing

Indexing and slicing of lists works just like in Strings. Let's make a new list to remind ourselves of how this works:

```
In [11]:
my_list = ['one','two','three',4,5,5,6,7,8,9,9,0]
```

```
In [3]:
# Grab element at index 0
my_list[0]
```

```
Out[3]: 'one'
```

```
In [12]:
# Grab index 1 and everything past it
my_list[1:10:2]
```

```
Out[12]: ['two', 4, 5, 7, 9]
```

```
In [8]:
# Grab everything UP TO index 3
my_list[:3]
```

```
Out[8]: ['one', 'two', 'three']
```

We can also use "+" to concatenate lists, just like we did for Strings.

```
In [9]:
my_list + ['new item',5]
```

```
Out[9]: ['one', 'two', 'three', 4, 5, 'new item', 5]
```

Note: This doesn't actually change the original list!

```
In [10]:
my_list
```

```
Out[10]: ['one', 'two', 'three', 4, 5]
```

In this case, you have to reassign the list to make the permanent change.

```
In [11]: # Reassign
         my_list = my_list + ['add new item permanently']
```

```
In [12]: my_list
```

```
Out[12]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

```
In [13]: l = [1,2,3,4]
```

```
In [14]: l * 2
```

```
Out[14]: [1, 2, 3, 4, 1, 2, 3, 4]
```

We can also use the * for a duplication method similar to strings:

```
In [13]: # Make the list double
         my_list * 2
```

```
Out[13]: ['one',
          'two',
          'three',
          4,
          5,
          'add new item permanently',
          'one',
          'two',
          'three',
          4,
          5,
          'add new item permanently']
```

```
In [14]: # Again doubling not permanent
         my_list
```

```
Out[14]: ['one', 'two', 'three', 4, 5, 'add new item permanently']
```

## Basic List Methods

If you are familiar with another programming language, start to draw parallels between lists in Python and arrays in other language. There are two reasons which tells why the lists in Python are more flexible than arrays in other programming language:

a. They have no fixed size (which means we need not to specify how big the list will be) b. They have no fixed type constraint

Let's go ahead and explore some more special methods for lists:

```
In [27]: # Create a new list
         l = [1,2,3]
```

Use the **append** method to permanently add an item to the end of a list:

```
In [28]:
```

```
# Append
l.append('append me!')
```

In [29]:
```
# Show
l
```

Out[29]:
```
[1, 2, 3, 'append me!']
```

Use **pop** to "pop off" an item from the list. By default pop takes off the last index, but you can also specify which index to pop off. Let's see an example:

In [18]:
```
# Pop off the 0 indexed item
l.pop(0)
```

Out[18]:
```
1
```

In [26]:
```
# Show
l
```

Out[26]:
```
[]
```

In [24]:
```
# Assign the popped element, remember default popped index is -1
popped_item = l.pop()
```

In [25]:
```
popped_item
```

Out[25]:
```
2
```

In [76]:
```
# Show remaining list
l
```

Out[76]:
```
[2, 3]
```

Note that lists indexing will return an error if there is no element at that index. For example:

In [30]:
```
l[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-30-e2a0c2623844> in <module>
----> 1 l[100]

IndexError: list index out of range
```

We can use the **sort** method and the **reverse** methods to also effect your lists:

In [31]:
```
new_list = ['a','e','x','b','c']
```

In [32]:
```
#Show
new_list
```

```
['a', 'e', 'x', 'b', 'c']
```

Out[32]:

In [34]:
```
# Use reverse to reverse order (this is permanent!)
new_list.reverse()
```

In [35]:
```
new_list
```

Out[35]:
```
['a', 'e', 'x', 'b', 'c']
```

In [82]:
```
# Use sort to sort the list (in this case alphabetical order, but for numbers it wil
new_list.sort()
```

In [83]:
```
new_list
```

Out[83]:
```
['a', 'b', 'c', 'e', 'x']
```

## Nesting Lists

Nesting Lists is one of the great features in Python data structures. Nesting Lists means we can have data structures within data structures.

For example: A list inside a list.

Let's see how Nesting lists works!

In [36]:
```
# Let's make three lists
lst_1=[1,2,3]
lst_2=[4,5,6]
lst_3=[7,8,9]

# Make a list of lists to form a matrix
matrix = [lst_1,lst_2,lst_3]
```

In [40]:
```
# Show
matrix
```

Out[40]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [39]:
```
matrix[2][1]
```

Out[39]:
```
8
```

In [ ]:

We can re-use indexing to grab elements, but now there are two levels for the index.

a. The items in the matrix object b. The items inside the list

In [86]:
```
# Grab first item in matrix object
matrix[0]
```

Out[86]:
```
[1, 2, 3]
```

In [87]:
```
# Grab first item of the first item in the matrix object
matrix[0][0]
```

Out[87]:
```
1
```

## List Comprehensions

Python has an advanced feature called list comprehensions which allows for quick construction of lists.

Before we try to understand list comprehensions completely we need to understand "for" loops.

So don't worry if you don't completely understand this section, and feel free to just skip it since we will return to this topic later.

Here are few of oue examples which helps you to understand list comprehensions.

In [43]:
```
# Build a list comprehension by deconstructing a for loop within a []
first_col = [row[0] for row in matrix]
```

In [44]:
```
matrix
```

Out[44]:
```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In [45]:
```
first_col
```

Out[45]:
```
[1, 4, 7]
```

## Advanced Lists

In this series of lectures, we will be diving a little deeper into all the available methods in a list object. These are just methods that should encountered without some additional exploring. Its pretty likely that you've already encountered some of these yourself!

Lets begin!

In [90]:
```
l = [1,2,3]
```

### append

Definitely, You have used this method by now, which merely appends an element to the end of a list:

In [46]:
```
l.append(4)
```

```
l
```

Out[46]:
```
[1, 2, 3, 'append me!', 4]
```

## count

We discussed this during the methods lectures, but here it is again. count() takes in an element and returns the number of times it occures in your list:

In [48]:
```
l.count(10)
```

Out[48]:
```
0
```

In [49]:
```
l.count(2)
```

Out[49]:
```
1
```

## extend

Many times people find the difference between extend and append to be unclear. So note that,

**append: Appends object at end**

In [52]:
```
x = [1, 2, 3]
x.append([4, 5])
print(x)
```

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

**extend: extends list by appending elements from the iterable**

In [53]:
```
x = [1, 2, 3]
x.extend('ty')
print(x)
```

```
[1, 2, 3, 't', 'y']
```

Note how extend append each element in that iterable. That is the key difference.

## index

index returns the element placed as an argument. Make a note that if the element is not in the list then it returns an error.

In [58]:
```
l.index('append me!')
```

Out[58]:
```
3
```

In [55]:
```
l
```

Out[55]:
```
[1, 2, 3, 'append me!', 4]
```

In [98]:
```
l.index(12)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-98-3f090052d656> in <module>()
----> 1 l.index(12)

ValueError: 12 is not in list
```

## insert

Two arguments can be placed in insert method.

Syntax: insert(index,object)

This method places the object at the index supplied. For example:

In [99]:
```
l
```

Out[99]:
```
[1, 2, 3, 4]
```

In [66]:
```
# Place a letter at the index 2
l.insert(100,'inserted')
```

In [70]:
```
l[100]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-70-e2a0c2623844> in <module>
----> 1 l[100]

IndexError: list index out of range
```

## pop

You most likely have already seen pop(), which allows us to "pop" off the last element of a list.

In [64]:
```
ele = l.pop(1)
```

In [65]:
```
l
```

Out[65]:
```
[1, 'inserted', 3]
```

In [104…
```
ele
```

Out[104…
```
4
```

## remove

The remove() method removes the first occurrence of a value. For example:

```
In [105…  l
Out[105…  [1, 2, 'inserted', 3]
```

```
In [106…  l.remove('inserted')
```

```
In [107…  l
Out[107…  [1, 2, 3]
```

```
In [108…  l = [1,2,3,4,3]
```

```
In [109…  l.remove(3)
```

```
In [110…  l
Out[110…  [1, 2, 4, 3]
```

## reverse

As the name suggests, reverse() helps you to reverse a list. Note this occurs in place! Meaning it effects your list permanently.

```
In [111…  l.reverse()
```

```
In [112…  l
Out[112…  [3, 4, 2, 1]
```

## sort

sort will sort your list in place:

```
In [113…  l
Out[113…  [3, 4, 2, 1]
```

```
In [114…  l.sort()
```

```
In [115…  l
Out[115…  [1, 2, 3, 4]
```

## Add element in list

1. append
2. extend
3. insert

```
In [15]: new_list = [1,2,3,4,5]
```

```
In [17]: len(new_list)
```

```
Out[17]: 5
```

```
In [18]: new_list.append(6)
```

```
In [19]: new_list
```

```
Out[19]: [1, 2, 3, 4, 5, 6]
```

```
In [20]: new_list.insert(3,7)
```

```
In [21]: new_list
```

```
Out[21]: [1, 2, 3, 7, 4, 5, 6]
```

## Remove element in list

1. remove
2. pop
3. del

```
In [30]: l = [1,2,3,4,5,6,2]
```

```
In [31]: l.remove(2)
```

```
In [32]: l
```

```
Out[32]: [1, 3, 4, 5, 6, 2]
```

```
In [33]: a = l.pop(1)
```

```
In [34]: a
```

```
Out[34]: 3
```

```
In [35]: del l[1]

In [36]: l

Out[36]: [1, 5, 6, 2]

In [37]: del l

In [39]: l

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
C:\Users\AN2025~1\AppData\Local\Temp/ipykernel_21664/3723490112.py in <module>
----> 1 l

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

## Reverse and Sort

```
In [51]: l = [5,62,34,4,5]

In [43]: l.reverse()

In [46]: l

Out[46]: [5, 62, 34, 4, 5]

In [58]: l.sort(reverse=True)

In [50]: l

Out[50]: [62, 34, 5, 5, 4]

In [62]: sorted(l,reverse=True)

Out[62]: [62, 34, 5, 5, 4]

In [61]: l

Out[61]: [62, 34, 5, 5, 4]

In [57]: a

Out[57]: [4, 5, 5, 34, 62]
```

## List has multiple references

```
In [63]: ac = [1,2,3,4,5,6]
```

```
In [64]: xyz = ac
```

```
In [65]: xyz
```

```
Out[65]: [1, 2, 3, 4, 5, 6]
```

```
In [66]: xyz.append(7)
```

```
In [67]: xyz
```

```
Out[67]: [1, 2, 3, 4, 5, 6, 7]
```

```
In [68]: ac
```

```
Out[68]: [1, 2, 3, 4, 5, 6, 7]
```

## String to List

```
In [72]: a = "one,two,three,four,five"
```

```
In [73]: list(a)
```

```
Out[73]: ['o',
 'n',
 'e',
 ',',
 't',
 'w',
 'o',
 ',',
 't',
 'h',
 'r',
 'e',
 'e',
 ',',
 'f',
 'o',
 'u',
 'r',
 ',',
 'f',
 'i',
 'v',
 'e']
```

```
In [76]:
```

```
b = a.split(",")
```

In [77]:
```
b
```

Out[77]:
```
['one', 'two', 'three', 'four', 'five']
```

In [79]:
```
",".join(b)
```

Out[79]:
```
'one,two,three,four,five'
```

## Count Method

In [83]:
```
l = [1,2,3,3,3,4,5,6,6,7]
```

In [81]:
```
l.count(3)
```

Out[81]:
```
3
```

In [84]:
```
l.count(6)
```

Out[84]:
```
2
```

## Indexing and Slicing

In [87]:
```
l[0]
```

Out[87]:
```
1
```

In [88]:
```
l[1]
```

Out[88]:
```
2
```

In [89]:
```
l[:]   #All elements
```

Out[89]:
```
[1, 2, 3, 3, 3, 4, 5, 6, 6, 7]
```

In [91]:
```
l[1:5]  # 2nd Element upto 5th Element, i.e index 1th element to index 4th Element
```

Out[91]:
```
[2, 3, 3, 3]
```

In [92]:
```
l[:5] #start element upto 4th index element
```

Out[92]:
```
[1, 2, 3, 3, 3]
```

In [93]:
```
l [1:] #1st index element upto last Element
```

```
Out[93]: [2, 3, 3, 3, 4, 5, 6, 6, 7]
```

```
In [94]: l[::2] #Start element upto last element skipping alternate elements
```

```
Out[94]: [1, 3, 3, 5, 6]
```

```
In [95]: l[::3] ##Start element upto last element skipping two elements
```

```
Out[95]: [1, 3, 5, 7]
```

## List Comprehension

```
In [ ]:
```

```
In [ ]:
```