# Module 2 Part 2: Python Strings and Lists

This module consists of 3 parts:

- **Part 1** - Introduction to Python.
- **Part 2** - Python Strings and Lists.
- **Part 3** - Python Tuples, Dictionaries, Reading data from a file, Formatting print output.

Each part is in a separate notebook. It is recommended to follow the order of the notebooks from Part 1 to Part 3.

# Table of Contents

# Strings

A **string** is a sequence of characters. In Python, the type of a string is **str**. Strings are enclosed with either single (') or double (") quotes. Alternatively, triple quotes (''') can be used to enclose strings that span multiple lines. Strings can contain letters, numbers or special characters. They can also be empty, i.e. `""`, or contain a spaces, e.g. `" "`.

A string can also be viewed an ordered sequence of characters. Individual characters in a string can be accessed using the character's index with the first character having an index value of 0.

A string's characters can also be accessed with a negative index. Negative indices count from the right-hand side with the final character at index -1, the second last at index -2 and so on.

```
In [1]: s1 = "Python is the Number 1 choice for social media hacking"
        print(s1[0])
        print(s1[1])
        s1[-1]
```

```
P
y
Out[1]: 'g'
```

Python provides many useful methods (or built-in functions) to work with strings. One of them is `len()`, which returns the number of characters in a string.

```
In [2]: len(s1)
```

```
Out[2]: 54
```

## Slicing

A *slice* is a segment of a string, beginning with a *start* index and continuing up to but **not including** an *end* index.

```
In [3]: s1[2:6]
```

```
Out[3]: 'thon'
```

The end index can be omitted in which case it defaults to `len(str)`. For example, `s1[2:]` is equivalent to the `s1[2:len(s1)]` substring from index 2 to the end of the string. Similarly, if the start index is omitted the string will be sliced from index 0.

```
In [4]: s1[:2]
```

```
Out[4]: 'Py'
```

In Python, strings are **immutable** - once created, strings cannot be changed. Instead, a new string must be created, which may contain a slice or slices of the existing string. For example, the string `'get'` cannot be changed to `'got'`. Similarly, the string `"I am learning Python"` cannot be changed to `"I know Python"` via an in-place change.

```
In [5]: s2 = "get"
        s2[1] = "o"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 2
      1 s2 = "get"
----> 2 s2[1] = "o"

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

In [ ]:
```
s2.replace('e', 'o')

# Note, that this does not change the s2 variable.
# Instead, a new string object is returned and can be assigned to a new variable.
```

In [ ]:
```
s3 = s2[0] + "o" + s2[2]
print(s3)

# Note that s2 was not modified'''

print(s2)

# Similarly
learn_str = "I am learning Python"
learn_str[:2] + "know" + " " + learn_str[14:]
```

## String operators

In the last example, the '+' operator was used. This operator, when used between two strings, **concatenates** them: `"abc123" + "def456"` returns `"abc123def456"`. A string can also be multiplied by an integer: `str * int` will concatenate `int` number of copies of `str`. However, a string cannot be multiplied by another string and all other mathematical operators will result in a `TypeError`.

Strings can be **compared** using the equality ( `==` ) and inequality ( `!=` ) operators. Operators `'>'` and `'<'` compare two strings by alphabetical or dictionary order, not by length. `'a' < 'b'` returns `True` and `'aaaaa' < 'bbb'` is also `True`. The capital letters are "less" than lowercase letters.

The operator `in` returns `True` if a substring appears anywhere inside a string.

In [6]:
```
print('a'<'A')
print('b' > 'aaaaa')
'on' in 'Python'
```

```
False
True
```

Out[6]:
```
True
```

We have seen the method `len()` above. To find other methods for strings, the function `dir(str)` can be used. This function will output a list of all methods available:

In [7]:
```
'''print() function is used for a better printout formatting'''

print(dir(str))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq_
_', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__
getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__l
e__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '_
_str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalp
ha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isp
rintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'mak
etrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'str
ip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

To learn more about a particular function, the `help` function can be used or a '?' can be typed before the name of the function.

For example, to get more information about the `find` method, you can call the `help()` function: `help(str.find)` or type `?str.find`. Here, `str` indicates that we are looking for the method applicable to string objects.

In [8]:
```
help(str.find)
```

```
Help on method_descriptor:

find(...)
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within S[start:end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.
```

**NOTE:** Square brackets around the description of the function indicate optional arguments. In the description of the `find` method, the start and end indices are optional. Moreover, this function can be called with the start index only, omitting the end index, as indicated by the nested square brackets.

## Traversing a string

Often we need to access all the items in a string exactly once each. The `for` loop allows for compact traversal structures. In the example below, the `for` loop selects each character in string `s1` and executes the body of the loop for each character.

In [ ]:
```
'''Traversing a string'''
```

```
for item in s1:
    if item in 'aeiouAEIOU':
        print(item)
```

In [ ]:
```
help(str.split)
```

## EXERCISE 2: Index of the second occurrence

The method `find` can be used in an expression like `s1.find(s4)` to find the index of the first occurrence of the substring `s4` in the string `s1`.

This method also takes an optional start argument, and the expression `s1.find(s4, start)` will return the first occurrence of the `s4` substring after the start index.

Write an expression to produce the second occurrence of the `s4` substring in the `s1` string.

In [10]:
```
# Let's check s1 and define s4:
print(s1)
s4 = 'ia'
print(s4)
```

```
Python is the Number 1 choice for social media hacking
ia
```

In [21]:
```
'''Type your code here'''

first = s1.find(s4)
s1.find(s4,first+1)
```

Out[21]:
```
44
```

In [ ]:
```
'''Exercise 2 solution:'''

'''First, let's find the first occurrence of the s4 substring.'''

first_occurrence = s1.find(s4)
first_occurrence
```

In [ ]:
```
'''Next, to find the second occurrence we should start after the first occurrence.'

s1.find(s4, first_occurrence + 1)
```

In [ ]:
```
'''Or these two expressions can be combined into one.'''

s1.find(s4, s1.find(s4)+1)
```

# Lists

A **list** is a sequence of values similar to a string. If a string is a sequence of characters, a list is a sequence of values of any type. The values in a list are called **elements** or **items**. Lists are versatile, and are sometimes called the "workhorses" of Python.

Lists can even contain other lists as elements, resulting in nested lists. The easiest way to create a list is to enclose a sequence of values in square brackets `[ ]`. Just like strings, lists can be empty. Working with lists is similar to working with strings. For example, the `len()` method will return the length of the list. Indexing and slicing of a list is also similar to strings.

In [12]:
```
'''List examples'''

empty_list = []
list_of_numbers = [1, 2.3, 4.5, 6]
list_of_strings = [""]
```

In [13]:
```
'''A string can be split into a list with the `split` method.
For the string 's1' defined above, a space (" ") can be used to split the string in

s1.split(" ")
```

Out[13]:
```
['Python',
 'is',
 'the',
 'Number',
 '1',
 'choice',
 'for',
 'social',
 'media',
 'hacking']
```

In [14]:
```
'''Please uncomment the line below and execute this command to see a list
of all methods available for a list object.'''

dir(list)
```

```
Out[14]: ['__add__',
          '__class__',
          '__class_getitem__',
          '__contains__',
          '__delattr__',
          '__delitem__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattribute__',
          '__getitem__',
          '__getstate__',
          '__gt__',
          '__hash__',
          '__iadd__',
          '__imul__',
          '__init__',
          '__init_subclass__',
          '__iter__',
          '__le__',
          '__len__',
          '__lt__',
          '__mul__',
          '__ne__',
          '__new__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__reversed__',
          '__rmul__',
          '__setattr__',
          '__setitem__',
          '__sizeof__',
          '__str__',
          '__subclasshook__',
          'append',
          'clear',
          'copy',
          'count',
          'extend',
          'index',
          'insert',
          'pop',
          'remove',
          'reverse',
          'sort']
```

Unlike strings, lists are **mutable**. Any item in a list can be changed; see example below:

```
In [15]: list_of_numbers[2] = 100
         list_of_numbers
```

```
Out[15]: [1, 2.3, 100, 6]
```

Python provides several methods that modify lists:

| **method** | **Description** |
|---|---|
| list.append(item) | Append an `item` to the end of the list. |
| list.extend(list_1) | Append the items in `list_1` to the list. |
| list.pop([index]) | Remove the item at `index` from the list, or the last item in the list if an index is not given. |
| list.remove(item) | Remove the first occurrence of the `item` . |
| list.reverse( ) | Reverse the list. |
| list.insert(int, item) | Insert an `item` at the given index `int` . The subsequent items in the list are shifted to the right. |

```
In [16]: list1 = [1,2,3,4]
         list1.append(5)
         list1.sort(reverse=True)
         list1
```

```
Out[16]: [5, 4, 3, 2, 1]
```

```
In [17]: list1.remove(3)
```

```
In [18]: list1
```

```
Out[18]: [5, 4, 2, 1]
```

```
In [19]: list2 = [11, 12]
         list1.extend(list2)
         list1
```

```
Out[19]: [5, 4, 2, 1, 11, 12]
```

## Traversing a list

A `for` loop can be used to access each item in a list, one at a time.

The structure in code is:

```
for elem in list_A:
    ...
```

A list can also be traversed using indices. This can be achieved with two built-in functions: `len()` and `range()` .

The documentation for `range()` states the syntax is as follows (Python Software Foundation, 2018):

```
range([start,] stop[, step])
```

The `range()` function returns a virtual sequence of numbers from `start` to `stop`, in intervals of the `step`.

Thus, the `range()` function is useful for iteration over a sequence of numbers. The start and step parameters are optional — calling the function with only one parameter will return a sequence from 0 up to but not including the stop value.

- `range(5)` returns the sequence 0, 1, 2, 3, and 4;
- `range(1,4)` returns 1, 2, 3;
- `range(1, 10, 3)` returns 1, 4, 7.

Combining `range()` and `len()` functions we can build a structure looping over a list by indices:

```
for i in range(len(list)):
    ...
```

## EXERCISE 3: Lists

1). Generate a list of integers from 0 to 5, reverse the list and print it out. Explain how you reversed the list.
**Hint:** The `append()` function will help you to generate the list by adding one integer to the list at a time.

In [36]:
```
'''Type your code here'''


list1 = [0,1,2,3,4,5]
print(list1)
list1.sort(reverse=True)
print(list1)

#I used the reverse function to reverse the list as shown in earlier in the module

list2 = []
for i in range(0,6):
    list2.append(i)
print(list2)

reverse = reversed(list2)
print(list(reverse))
```

```
[0, 1, 2, 3, 4, 5]
[5, 4, 3, 2, 1, 0]
[0, 1, 2, 3, 4, 5]
[5, 4, 3, 2, 1, 0]
```

2). Generate a list of strings. Add the items from the list of strings to the list of integers from step 1. Explain your choice of string method.

In [40]:
```
'''Type your code here'''

string_list = ["apple", "pear", "orange"]
string_list


list1.extend(string_list)
list1
```

Out[40]:
```
[5, 4, 3, 2, 1, 0, 'apple', 'pear', 'orange']
```

In [ ]:
```
'''Exercise 3 Solution:'''

'''1). An empty list can be created first.'''
list_integers = []

'''Than integers can be added to the list.'''
for i in range(0,6):
    list_integers.append(i)

'''Printing out the whole list:'''
print(list_integers)

'''or by elements:'''
for n in list_integers:
    print(n)
```

In [ ]:
```
'''1(a). To reverse the list, the 'reversed()' function can be used.
This function returns an iterator, which accesses the list in reversed order.'''
rev_iterator = reversed(list_integers)

'''To return the list we need to add list() in front.'''
print(list(rev_iterator))

'''These operations did not change our original list:'''
print(list_integers)
```

In [ ]:
```
'''1(b). Another way is to slice the list to reverse it.
Syntax [start:stop:step]'''
reversed_list_2 = list_integers[::-1]

'''To make a copy of the whole list the start and stop are omitted.
Here negative step will slice over the list in reverse.'''
print('Reversed list:')
print(reversed_list_2)

'''Slice does not modify the original list, it returns a new list.'''
```

```
print('This is the original list:')
print(list_integers)
```

In [ ]:
```
'''1(c). The 'reverse()' function will reverse the list in place:'''
list_integers.reverse()

print ("Using reverse() function:")
print (list_integers)

'''The original list is reversed'''
```

In [ ]:
```
'''2). A list of strings is created:'''

sequences = ['string', 'list']
```

In [ ]:
```
'''2(a). The method 'extend()' takes a list and appends all the elements from the s
not the second list itself.'''

list_integers.extend(sequences)
list_integers
```

In [ ]:
```
'''2(b). The method 'append()' adds an object to the end of a list:'''

list_integers.append(sequences)
list_integers
```

## List comprehension

**List comprehension** in Python provides an easy and elegant way of creating a new list. A common application of list comprehension is to create a new list based on an existing sequence(s), e.g. one or more lists. While creating a new list with list comprehension, we might want to include only elements that satisfy a certain condition, or transform elements of an original list using an operation, or a calculation, applied to these elements.

For example, we might have a list of numbers, `nums`. We need to create a new list, let's call it `squares`, where each element is a square of the corresponding element from the list `nums`. We can write a `for` loop which will look as follows:

In [41]:
```
nums = [0, 1, 2, 3, 4, 5, 6, 7]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49]
```

However, a more simple and elegant way would be to write the same loop in one line of code using list comprehension:

In [42]:
```
squares = [x ** 2 for x in nums]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49]
```

In [43]:
```
'''Another example: create a new list from 2 lists,
where the condition is - use only those numbers that are common in both lists, list

listA = [15, 35, 76, 83, 910, 1234]
listB = [1234, 234, 83, 3, 4, 5]

new_list = []

for a in listA:
    for b in listB:
        if a == b:
            new_list.append(a)

print(new_list)
```

```
[83, 1234]
```

In [44]:
```
'''The same result achieved using list comprehension:'''

[a for a in listA for b in listB if a == b]
```

Out[44]:
```
[83, 1234]
```

In [45]:
```
'''Another example:
return a list of doubled numbers only if the number is an odd number'''

[n * 2 for n in listA if n % 2 == 1]
```

Out[45]:
```
[30, 70, 166]
```

The general structure of a list comprehension can be written as follows:

```
[<output expression> <loop expression <input expression>> <optional
predicate expression>]
```

A list comprehension is always enclosed in brackets. It starts with an expression followed by a `for` expression, then zero or more `for` or `if` clauses.

The list comprehension can return a list of pairs, or tuples. We will learn about tuples in the next section of this module. For now, and to wrap up the discussion about list comprehension, here is how to create a list of numbers paired with their square:

In [ ]:
```
[(x, x**2) for x in range(6)]
```

**End of Part 2**

This notebook makes up one part of this module. Now that you have completed this part, please proceed to the next notebook in this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

## References

McKinney, W. (2017). Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython (pp. 15-50). O'Reilly Media.

Python Software Foundation, (2018). Built-in Functions. Retrieved from (https://docs.python.org/3/library/functions.html#func-range).