7 | PYTHON COLLECTIONS

Introduction

Previous notebooks showed the creation and assignment of single values such as numbers and strings to variables. Python allows for the combination of such elements into collections.

There are four types of collections. These are lists, tuples, dictionaries, and sets. Understanding these collection types is not just about learning Python syntax. It is about the tools used to handle complex, real-world problems that revolve around data. As the old adage goes, *data is the new oil*, and Python collections are the pipelines used to transport and process this data.

Lists

A Python **list** is a built-in data structure that allows for the storage of a collection of items. The items, known as elements, can be of any type. Integers, strings, floats, or even other complex objects, such as other lists, dictionaries, and custom objects. This makes lists incredibly versatile and adaptable to a wide variety of tasks.

A Python list can be created using square bracket notation. The elements are separated by commas. The following code creates a list of integers and uses the `type` function to confirm the data type of a list.

In [1]:
```
# Create a list of five integers
[1, 2, 3, 4, 5]
```

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

In [2]:
```
# Type of the list
type([1, 2, 3, 4, 5])
```

Out[2]:
```
list
```

The elements can alse be of different types. The following code creates a list of integers and strings.

In [3]:
```
# Create a list of five integers and five strings
[1, 2, 3, 4, 5, "one", "two", "three", "four", "five"]
```

Out[3]:
```
[1, 2, 3, 4, 5, 'one', 'two', 'three', 'four', 'five']
```

Python lists are ordered, meaning the order in which elements are added to the list is preserved. Each element in a list has a specific position, identified by its index. Indexing in Python is zero-based, which means the first element is at index 0, the second element is at index 1, and so forth. Negative indexing is also supported, where the last element is at index -1, the second last at -2, and so on, making it possible to access the list from both ends.

The index value can be used to retrieve an element from a list. The list object below is assigned to the variable `my_list`. The first element in the list is retrieved using the index value 0.

In [4]:
```
# Create a lsit of five random integers and assign it to the variable my_lis
my_list = [7, 4, 8, 2, 9]
```

In [5]:
```
# Retrieve the first element of the list
my_list[0]
```

Out[5]: 7

The first element is indeed 7. The last element can be retrieved using the index value -1. This is useful when the length of the list is not known and particularly long.

In [6]:
```
# Retrieve the last element of the list
my_list[-1]
```

Out[6]: 9

Note that in this short list, it is easy to recognize that the last element has an index of 4.

In [7]:
```
# Retrieve the last element of the list
my_list[4]
```

Out[7]: 9

Contiguous elements can be retrieved using a range of index values. This process is termed slicing. The following code retrieves the first three elements of the list. The first three elements have indices 0, 1, and 2.

In [8]:
```
# Retrieve the first three elements of the list
my_list[0:3]
```

Out[8]: [7, 4, 8]

While the range is specified as $0:3$, the element at index 3 is not included in the result. This is because the range is exclusive of the last index value.

The colon symbol, `:`, can be used to retrieve all elements in a list up to a specific index value. The following code retrieves all elements up to index 3.

In [9]:
```
# Retrieve the first three elements of the list
my_list[:3]
```

Out[9]:
```
[7, 4, 8]
```

The same process can be used to retrieve all elements from a specific index value to the end of the list. The following code retrieves all elements from index 3 to the end of the list.

In [10]:
```
# Retrieve all elements from index 3
my_list[3:]
```

Out[10]:
```
[2, 9]
```

In [11]:
```
# Retrieve the last three elements of the list
my_list[-3:]
```

Out[11]:
```
[8, 2, 9]
```

One of the key characteristics of a Python list is its mutability. In other words, once a list is created, its elements can be modified, added, or removed, allowing for dynamic and flexible data manipulation. Functions like `append`, `extend`, and `pop`, among others, provide the capability to modify lists after they have been created.

The index is specified before assigning the new value. The following code changes the first element in the list to the string `'first'`.

In [12]:
```
# Change the first element of the list to 'First
my_list[0] = 'First'
my_list
```

Out[12]:
```
['First', 4, 8, 2, 9]
```

The code below demonstrates reassignment with anotehr list to show that lists can also be elements of a list object. Such lists of lists are known as nested lists.

In [13]:
```
# Change the last element of the list to the list ['one', 'two', 'three']
my_list[-1] = ['one', 'two', 'three']
my_list
```

Out[13]:
```
['First', 4, 8, 2, ['one', 'two', 'three']]
```

It is clear from using the `len` function that the list still has only five elements, even though the last element is a nested list with three elements.

In [14]:
```
# Return the length of the list
len(my_list)
```

Out[14]:
```
5
```

The nested list can be accessed using the index of the outer list.

In [15]:
```
# Retrieve the last element of the list
my_list[-1]
```

Out[15]:
```
['one', 'two', 'three']
```

To retrieve elements in a nested list, the index of the outer list is specified first, followed by the index of the inner list. The following code retrieves the first element of the nested list.

In [16]:
```
# Retrieve the first element of the last element of the list
my_list[-1][0]
```

Out[16]:
```
'one'
```

The `append` method adds an element to the end of a list. The following code adds the integer 6 to the end of the list.

The `extend` method can be used to add elements to a list as well. The following code adds the elements 10, 11, and 12 to `my_list`. Note that the values are passed as a list.

In [17]:
```
# Extend the list with the elements 10, 11, and 12
my_list.extend([10, 11, 12])
my_list
```

Out[17]:
```
['First', 4, 8, 2, ['one', 'two', 'three'], 10, 11, 12]
```

The `pop` method removes the last element from a list. The following code removes the last element, which is the integer 12, from `my_list`.

In [18]:
```
# Remove the last element of the list
my_list.pop()
my_list
```

Out[18]:
```
['First', 4, 8, 2, ['one', 'two', 'three'], 10, 11]
```

By passing an index value to the `pop` method, a specific element can be removed from a list. The following code removes the element at index 2 from `my_list`.

In [19]:
```
# Remove the element at index 2
my_list.pop(2) # The value 8 at index number 2 is removed
my_list
```

Out[19]:
```
['First', 4, 2, ['one', 'two', 'three'], 10, 11]
```

In [20]:
```
# Append the number 6 to the list
my_list.append(6)
my_list
```

Out[20]:
```
['First', 4, 2, ['one', 'two', 'three'], 10, 11, 6]
```

The `insert` method adds an element to a list at a specific index. The following code adds the string `'inserted'` to `my_list` at index 2.

In [21]:
```
# Insert the string 'inserted' at index 2
my_list.insert(2, 'inserted')
my_list
```

Out[21]:
```
['First', 4, 'inserted', 2, ['one', 'two', 'three'], 10, 11, 6]
```

In addition to this, Python lists support operations like concatenation, repetition, and membership tests. Two new list are created below and the addigion symbol, `+`, is used to concatenate them.

In [22]:
```
# Create a two new lists of integers
my_list1 = [1, 2, 3, 4, 5]
my_list2 = [6, 7, 8, 9, 10]
```

In [23]:
```
# Concatenate the two lists
my_list1 + my_list2
```

Out[23]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

The elements of `my_list1` is repeated three times using the multiplication symbol, `*`, below.

In [24]:
```
# Repeat my_list1 three times
my_list1 * 3
```

Out[24]:
```
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

The `in` keyword is used to determine of a specified element is contained in a list. Below it is used below to determine if the integer 3 is an element of `my_list1`.

In [25]:
```
# Determine of the numner 5 is in my_list1
5 in my_list1
```

Out[25]:
```
True
```

The `reverse` method revereses the order of the elements of a list object.

In [26]:
```
# Reverse the list
my_list1.reverse()
```

```
my_list1
```

Out[26]:
```
[5, 4, 3, 2, 1]
```

Instead of using bracket notation, the `list` function can be used to create a list. The following code creates a list of integers from 0 to 9 using the `range` function.

In [27]:
```
# Use the list function and the range function to create a list of integers
list(range(10))
```

Out[27]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Passing a string to the `list` function, uses the individual characters of the string as elements of the list.

In [28]:
```
# Created a list containing the string 'I love Python'
list('I love Python')
```

Out[28]:
```
['I', ' ', 'l', 'o', 'v', 'e', ' ', 'P', 'y', 't', 'h', 'o', 'n']
```

Python lists are typically used when there's a need for a collection of data that will undergo modification during the course of a program, such as adding, removing, or changing elements. They form a foundational part of Python programming and offer a flexible and intuitive way to handle ordered collections of data.

Tuples

A tuple is similar to a list, other than the fact that tuples are immutable. The tuple or its elements cannot be changed after being created.

Parentheses are used instead of square brackets to denote a tuple (although they are not strictly required). The `tuple` function similarly generates a tuple object.

The code below generates a tuple, assigning elements separated by commas. No parentheses are used.

In [29]:
```
# Create a tuple of five integers and assign it to the variable tuple_1
tuple_1 = 2, 4, 5, 7, 'ring' # Simply state elements
tuple_1
```

Out[29]:
```
(2, 4, 5, 7, 'ring')
```

The `type` function indicates that `tuple_1`` is indeed a tuple object.

In [30]:
```
# Type of the tuple
type(tuple_1)
```

Out[30]:
```
tuple
```

The elements of a tuple can be of many data types.

In [31]:
```
# Create a tuple with the elements 1, 2, 'three', and the list 'Python' and
tuple_2 = (1, 2, 'three', list('Python')) # Use parentheses
tuple_2
```

Out[31]:
```
(1, 2, 'three', ['P', 'y', 't', 'h', 'o', 'n'])
```

Indexing and slicing of tuples is similar to that of lists. The following code retrieves the first element of `tuple_1`.

In [32]:
```
# Retrieve the first element of tuple_1
tuple_1[0]
```

Out[32]:
```
2
```

Since tuples are immutable, the values of elemenst cannot be changed. Nor can elements be added or removed from a tuple.

Other than using the indices for each element of a tuple, a computer variable name can be assigned to each element of a tuple. The code below generates a tuple object, `languages``. The tuple contains three string objects.

In [33]:
```
# Create a tuple with the string elements 'Python', 'R', and 'Julia' and ass
languages = ('Python', 'R', 'Julia')
languages
```

Out[33]:
```
('Python', 'R', 'Julia')
```

Each element is assigned a variable name.

In [34]:
```
# Assign each element of the tuple to a separate variable, using the variabl
one, two, three = languages
```

Using the variable names, the elements of the tuple can be retrieved.

In [35]:
```
# Retrieve the object in one
one
```

Out[35]:
```
'Python'
```

The `zip` function is an interesting and useful function. It combines elements of lists into a list of tuples. The code below generates two list object. In the code below, the first, `names`, contains three first names. The second, `surnames` contains a list of three last names.

In [36]:
```
names = ['Mary', 'Anne', 'Rene']
surnames = ['Johnstone', 'Barnes', 'Le Roux']
```

The `zip` function is used by passing the two list objects as arguments. All of these are passed as an argument to the ``list`` function. The result is a list of tuples, where each element of a tuple comes from an ordered pairing of the original list objects. Note therefor, that the original list objects must have the same length (the same number of elements).

In [37]:
```
# Use the zip function to create a list of tuples
list(zip(names, surnames))
```

Out[37]:
```
[('Mary', 'Johnstone'), ('Anne', 'Barnes'), ('Rene', 'Le Roux')]
```

A list of tuples can be unzipped using `*` notation.

In [38]:
```
# Generate a list of tuples (each with three elements)
data_set = [('Anna', 'Rogers', 33), ('Susan', 'Roberts', 25), ('Rene', 'Du B

# Unzip the thee element tuples
names, lastnames, ages = zip(*data_set)
```

In [39]:
```
# Return the names
names
```

Out[39]:
```
('Anna', 'Susan', 'Rene')
```

In [40]:
```
# Return the lastnames
lastnames
```

Out[40]:
```
('Rogers', 'Roberts', 'Du Bois')
```

In [41]:
```
# Return the ages
ages
```

Out[41]:
```
(33, 25, 28)
```

Task: Create a tuple named `primes` containing the first seven prime numbers. Then use indexing to return the last three primes numbers in the tuple.

Dictionaries

Dictionaries are key-value pairs. They are very useful and powerful Python collections and form the basis of many data structures in Python.

The `dict` function creates a dictionary. It is much more common to see the use of curly braces, though. The key and the value are separated by a colon, and each key-value pair is separated by a comma.

The code cell below is used to generate data for a subject in a clinical research project. The variable name is chosen as the study protocol identity of each participant.

In [42]:
```
# Generate a dictionary
id345 = {'First Name':'Jenny', 'Last Name':'Gregory', 'Age':48, 'Heart Rate'
```

The `keys` method returns a `dict_keys` object that contains a list of the keys in a dictionary.

In [43]:
```
# Return the keys of the dictionary
id345.keys()
```

Out[43]:
```
dict_keys(['First Name', 'Last Name', 'Age', 'Heart Rate'])
```

The `values` method returns a `dict_values` object that contains alist of all the values.

In [44]:
```
# Return the values of the dictionary
id345.values()
```

Out[44]:
```
dict_values(['Jenny', 'Gregory', 48, 75])
```

The key can be used as an index to retrieve a value.

In [45]:
```
# Return the value for the 'Firt Name' key
id345['First Name']
```

Out[45]:
```
'Jenny'
```

The `get` method can also be used. Where the index notation returns an error if a key is not found, the `get` method returns nothing instead of an error message.

In [46]:
```
id345.get('Surname') # No such key
```

Task: Generate a Python dictionary object assigned to the variable `my_dict`, that contains the keys `1`, `2`, `3`, and `4` with the corresponing values `A`, `B`, `C` and `D`. Note how the keys can be numbers. Return a Python list object from `my_dict` that contains the values of the dictionary.

Solution

In [47]:
```
# Generate a dictionary using the dict function
my_dict = dict([(1, 'A'), (2, 'B'), (3,'C'), (4,'D')])

# Return a list of the values of the dictionary
list(my_dict.values())
```

Out[47]:
```
['A', 'B', 'C', 'D']
```

Sets

A Python set object is an unordered collection of unique elements. In this regard a Python set is much like a mathematical set.

A set is generated using curly brackets or the `set` function. The latter takes a list as argument.

The code below generates a set using braces and set notation. Note that in eacn case there is a duplication of elements. A set object only contains unique elements and discards the duplicates.

In [48]:
```
# Generate set with the elements 1, 2, 3, 4, 4, 4, 4, 1
set_1 = {1, 2, 3, 4, 4, 4, 4, 1}
set_1 # Call object and print to the scren
```

Out[48]:
```
{1, 2, 3, 4}
```

In [49]:
```
# Use the set function instead
set_1 = set([1, 2, 3, 4, 4, 4, 4, 1])
set_1
```

Out[49]:
```
{1, 2, 3, 4}
```

The `add` method adds an element to a set.

In [50]:
```
# Add the element 7 to the set
set_1.add(7)
set_1
```

Out[50]:
```
{1, 2, 3, 4, 7}
```

The `remove` method removes an element from a set. Note that it is not the index that is used.

In [51]:
```
# Remove the element 4 from the set
set_1.remove(4)
set_1
```

Out[51]:
```
{1, 2, 3, 7}
```

As with mathematical sets, the union and intersetcion of sets can be determined. The `union` method returns a set that contains all the elements of both sets and the `intersection` method returns a set that contains only the elements that are common to both sets.

In [52]:
```
# First set
set_A = {'a', 'a', 'a', 'b', 'c', 'd', 'e', 'f', 'e', 'a'}
```

```
set_A
```

Out[52]:
```
{'a', 'b', 'c', 'd', 'e', 'f'}
```

In [53]:
```
# Second set
set_B = {'a', 'c', 'c', 'f', 'g'}
set_B
```

Out[53]:
```
{'a', 'c', 'f', 'g'}
```

In [54]:
```
# Return the union of the two sets using the union method
set_A.union(set_B)
```

Out[54]:
```
{'a', 'b', 'c', 'd', 'e', 'f', 'g'}
```

In [55]:
```
# Return the intersection of the two sets using the intersection method
set_A.intersection(set_B)
```

Out[55]:
```
{'a', 'c', 'f'}
```

As alternative, the `|` and `&` operators can be used to determine the union and intersection of sets, respectively.

In [56]:
```
# Return the union of the two sets using the pipe operator
set_A | set_B
```

Out[56]:
```
{'a', 'b', 'c', 'd', 'e', 'f', 'g'}
```

In [57]:
```
# Return the intersection of the two sets using the & operator
set_A & set_B
```

Out[57]:
```
{'a', 'c', 'f'}
```

The set difference operation removes all the elements in the intersection of two sets from the first set. In this sense, it is similar to subtraction. The code below uses the `difference` method to calculate set difference.

In [58]:
```
# Return the difference between the two sets using the difference method
set_A.difference(set_B)
```

Out[58]:
```
{'b', 'd', 'e'}
```

Task: Reverse the order of the set difference in the code cell above and determine if the set difference between the objetcs `set_A` and `set_B` is the same as the set difference between the sets `set_B` and `set_A`.

The symmetric difference between sets subtracsts the intersection of the sets from the union of the sets. The code cell below uses the `symmetric_difference` method to calculate the set difference between two sets.

In [59]:
```
# Return the symmetric difference between the two sets using the symmetric_d
set_A.symmetric_difference(set_B)
```

Out[59]:
```
{'b', 'd', 'e', 'g'}
```

List comprehension

Python has a powerful feature called comprehension. It is most commonly used in terms of lists and is then termed **list comprehension**. Comprehension combines for loops into the creation of a collection.

Below, list comprehension is used to generate a list that contains the square of the first 10 natural numbers. Square bracket notation is used for list comprehension. The first expression is the recipe instruction. In this case, a placeholder variable, `x`. Then follows the `for` loop with the in keyword.

In [60]:
```
# Use list comprehension to create a list of the first 10 square natural num
[x**2 for x in range(1, 11)]
```

Out[60]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Conditionals can be used as well. Below, only the squares of the natural numbers greater than 5 is squared.

In [61]:
```
# Squares of natural numbers larger than 5
[n**2 for n in range(1, 11) if n > 5]
```

Out[61]:
```
[36, 49, 64, 81, 100]
```

The `if` statement can also contain a recipe. Below, only cases where the square is smaller than 50 is included.

In [62]:
```
# Only include elements that are less than 50
[n**2 for n in range(1, 11) if n**2 < 50]
```

Out[62]:
```
[1, 4, 9, 16, 25, 36, 49]
```

Quiz questions

Questions

1. How do you define a list in Python?
2. How do you access the third element of a Python list named `my_list`?

3. How can you add the element "new_item" to the end of an existing list named `my_list`?
4. *How do you create a Python tuple with the items "apple", "banana", and "cherry"?
5. How do you access the first element of a Python tuple named `my_tuple`?
6. How do you define a dictionary in Python with keys "name", "age" and values "John Doe", 30?
7. How can you retrieve the value for the key "name" from the dictionary `my_dict`
8. How can you change the value associated with the key "age" in the dictionary `my_dict` to 35?
9. How can you add a new key-value pair "city":"New York" to the dictionary `my_dict`?
10. *How do you create a set in Python with the items "apple", "banana", and "cherry"?
11. How do you add the item "orange" to an existing set `my_set`?
12. How do you remove the item "banana" from the set `my_set`?
13. What is the length of a list `my_list` with elements `[1, 2, 3, 4, 5]`?
14. What will be the result of `my_list * 2` if `my_list` is `[1, 2, 3]`?
15. What is the result of `set([1,2,2,3,4,4,4,5,5])`?