A2 Topic 4: Introduction to Lists

Sometimes you need to store more than one value in a variable. For example, the temperature for each day during a month. For this, you would need 30 variables, but to manage that many variables would be very tedious! The solution to this is to use a 'list', which lets you store all 30 values with just one variable. Consider the example where you need the average temperature for a week.

In [1]:

```
dayTemp1 = 22
dayTemp2 = 18
dayTemp3 = 20
dayTemp4 = 21
dayTemp5 = 23
dayTemp6 = 15
dayTemp7 = 17
averageTempWeek = (dayTemp1 + dayTemp2 + dayTemp3 + dayTemp4 + dayTemp5 + dayTemp6 + dayTe
mp7)/7
print("Average Temperature per week ", round(averageTempWeek,2)) #the average is round a t
wo decimals
```

Out[1]:

```
Average Temperature per week 19.43
```

You can manage that with separate variables for one week, but what if you wanted two weeks, three weeks, or more?

You can combine these temperatures into a single variable called 'temps' that stores a list as follows:

In [2]:

```
# to create a list, list the items to be included separated by commas inside of square bra
ckets
temps = [22, 18, 20, 21, 23, 15, 17]

print(temps)
```

Out[2]:

```
[22, 18, 20, 21, 23, 15, 17]
```

You can check the type of `temps` by using the `type()` function.

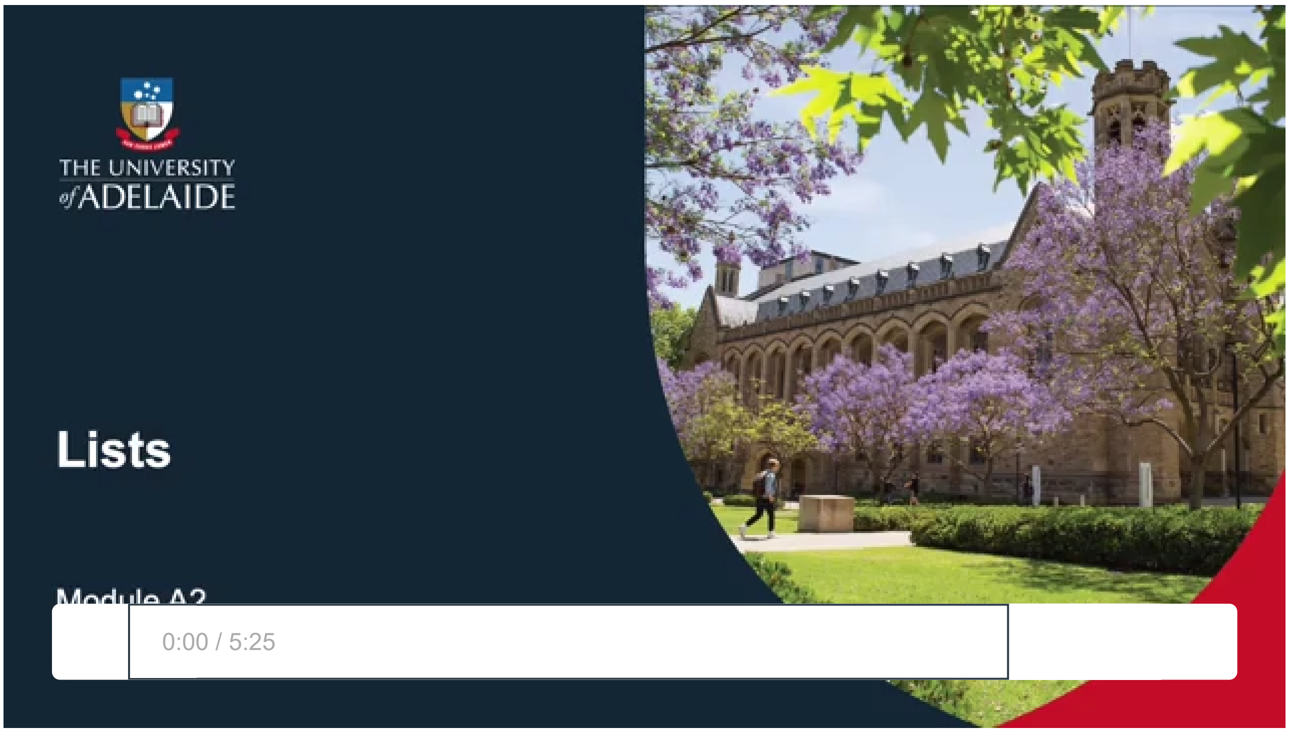
In [3]:

```
#Getting the type of temps
type(temps)
```

Out[3]:

```
<class 'list'>
```

Lists are one of four built-in data types in Python that we use to store collections of data. The other three data types are Dictionaries, Tuples and Sets that you will also learn in this module. The following video introduces what lists are in Python.



Lists are used to store multiple elements in a single variable. You can get the number of elements in the list by using the `len()` function.

In [4]:
```
temps = [22, 18, 20, 21, 23, 15, 17]
len(temps)
```

Out[4]:
```
7
```

Your turn for some practice: Putting Data Together

1. Write Python code to create a list using the following numbers. Print the list. Observe whether the order of the numbers has changed or duplicates have been removed. Then calculate and print the length of the list.

```
1, 2, 6, 10, 9, 6, 10, 1, 9, 1
```

Info: List Items

- List items are ordered, meaning that the items in a list have a defined order, and that order will not change.
- List also allows having duplicate values.

List types

To declare an empty list, you only need to assign a variable with square brackets, as shown in the example below.

In [5]:
```
#Assigning an empty list to the variable lst1
lst1 = []
print(lst1)
print(type(lst1))
print(len(lst1))
```

Out[5]:
```
[]
<class 'list'>
0
```

A 'list' is a container and can contain different data types and even mixed types of data.

In [6]:
```
#Example of a list with only numbers
lst1 = [ -1, 2, 5.5, 3.3]
print(lst1)
```

Out[6]:
```
[-1, 2, 5.5, 3.3]
```

In [7]:
```
#Example of a list with strings and numbers
listNumbers = ['zero', 1, 2.0, 3, 'four']
print (listNumbers)
```

Out[7]:
```
['zero', 1, 2.0, 3, 'four']
```

A list can contain even another list, which in turn can contain other lists, and so on. It is commonly known as a *nested list*. This allows you to arrange data into hierarchical structures.

In [8]:
```
#Example of a list with different types including another list
listNumbers = ['zero', 1, 2.0, 3, 'four']
listThings =[1, 'John', False, listNumbers ]
print(listThings)
```

Out[8]:
```
[1, 'John', False, ['zero', 1, 2.0, 3, 'four']]
```

In [9]:
```
listThings = ['John', [90, [False, 'Smith'], 19, True], 'Denise', 'David']
print(listThings)
print(type(listThings))
print(len(listThings))
```

Out[9]:
```
['John', [90, [False, 'Smith'], 19, True], 'Denise', 'David']
<class 'list'>
4
```

If you want to clearly see the hierarchical structures of these nested lists and display the contents in an indented tree-like formation, you can use pretty-printing using `pprint`. Try changing the `indent` value and observe what it does. Since `pprint` produces clear hierarchical views on collection of data such as *lists*, it is also used for debugging purposes (for situations where we use `print` statements).

In [10]:
```
from pprint import pprint
lst1 = ['John', 91, [90, [False, 'Smith'], 19, True, 'David'], 'Denise', 91, 'David', 'Dav
id', 91]
pprint(lst1, indent=1)
```

Out[10]:
```
['John',
 91,
 [90, [False, 'Smith'], 19, True, 'David'],
 'Denise',
 91,
 'David',
 'David',
 91]
```

Your turn for some practice: List Types

1. In your Jupyter notebook, create the three lists described below and print them:
    1. list1 - Including a sequence of integer numbers [-10, 10, 2], three float numbers and one string
    2. list2 - Including booleans (False, True), two strings and two numbers.
    3. list3 - Including list1, list2 and add any data type of your preference.

Next Steps

In this section, you learned that we use *lists* to store the sequence of various types of data. Now you know how to develop:

- an empty list
- list with multiple data types
- nested lists
- get the type and length of lists

Next, we will look at how we can perform basic operations using lists such as adding extra elements, updating the values in lists and deleting elements.

A2 Topic 4: List Operations

Python has many useful list operations that make it really easy to work with lists.

Lists Basic Operators

Lists have the same basic operators that strings have:

- `+` is used to concatenate lists

In [1]:
```
#Example of concatenation
lst1 = [1, 2.5, 'No', "Yes", 5]
lst2 = [False, True, "Python", 1.7]
lst3 = lst1 + lst2
print(lst3)
```

Out[1]:
```
[1, 2.5, 'No', 'Yes', 5, False, True, 'Python', 1.7]
```

- `*` is used to repeat a list

In [2]:
```
#Example of repetition
lst1 = [1, 2.5, 'No', "Yes", 5]
lst1 = lst1*2
print(lst1)
```

Out[2]:
```
[1, 2.5, 'No', 'Yes', 5, 1, 2.5, 'No', 'Yes', 5]
```

- `in` is used to determine if a specific element is in the list. Returns 'True' when the element is in the list and 'False' when not.

In [3]:
```
#Example of in
lst1 = ['banana', 'orange', 3, 5.5, 'white', 'watermelon']
'white' in lst1
```

Out[3]:
```
True
```

In [4]:
```
if 'white' in lst1:
    print('The color white is in the list')
```

Out[4]:
```
The color white is in the list
```

- `not in` is to check if a specific element **is not** in the list. Returns a boolean value.

In [5]:
```
#Example of not in
'red' not in lst1
```

Out[5]:
```
True
```

In [6]:
```
if 'red' not in lst1:
    print('The color red IS NOT in the list')
```

Out[6]:
```
The color red IS NOT in the list
```

Python has some list methods that you can use to manage a list efficiently.

Add List Elements

To add an element at the end of the list, you need to use `append(element)` function, as shown in the example below.

In [7]:
```
#Example of append
colors = ['black', 'white ', 'yellow']
print ('colors before append ', colors)
colors.append('red')
print ('colors after append  ', colors)
```

Out[7]:
```
colors before append  ['black', 'white ', 'yellow']
colors after append   ['black', 'white ', 'yellow', 'red']
```

To append elements from one list to another, you can use the `extend()` function. The difference between the `+` function (that we learnt earlier) and `extend()` is that the former creates a new list and the latter modifies an existing list. See the below example.

In [8]:
```
lst1 = [1, 2.5, 'No', "Yes", 5]
lst2 = [False, True, "Python", 1.7]
lst3 = lst1 + lst2
print(lst3)
lst1.extend(lst2)
print(lst1)
```

Out[8]:
```
[1, 2.5, 'No', 'Yes', 5, False, True, 'Python', 1.7]
[1, 2.5, 'No', 'Yes', 5, False, True, 'Python', 1.7]
```

Remove List Elements

To search and remove the first matching element in a list, you need to use `remove(element)` function. If the element is not in the list, the method returns an error.

In [9]:
```
#Example of remove
colors = ['black', 'white ', 'yellow', 'red']
print ('colors before remove ', colors)
colors.remove('yellow')
print ('colors after remove  ', colors)
```

Out[9]:
```
colors before remove  ['black', 'white ', 'yellow', 'green', 'red']
colors after remove   ['black', 'white ', 'green', 'red']
```

In [10]:
```
#What happens if we try to remove an element that is not present in the list
colors.remove('purple')
```

Out[10]:
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-42-89b5cbdd3cda> in <module>
      1 #What happens if we try to remeve an element that is not present in the list
----> 2 colors.remove('purple')

ValueError: list.remove(x): x not in list
```

If you use `pop()` function, it will remove the last element in the list.

In [11]:
```
colors = ['black', 'white ', 'yellow', 'red']
colors.pop()
print(colors)
```

Out[11]:
```
['black', 'white ', 'yellow']
```

What if you want to delete the entire list? You can use `del` keyword.

In [12]:
```
colors = ['black', 'white ', 'yellow', 'red']
del colors
print(colors)
```

Out[12]:
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
ipykernel.py in <module>
      1 colors = ['black', 'white ', 'yellow', 'red']
      2 del colors
----> 3 print(colors)

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

Instead of deleting the list completely, you can also empty the list by using `clear()` function.

In [13]:
```
colors = ['black', 'white ', 'yellow', 'red']
colors.clear()
print(colors)
```

Out[13]:
```
[]
```

Copy Lists

One of the most common problems programmers encounter is copying lists. When you assign a list (`lst1`) to another list (`lst2`), e.g. `lst2 = lst1`, both `lst2` and `lst1` refer to the **same** list. If you modify either of the lists, the other list is modified as well.

In [14]:
```
lst1 = ['black', 'white ', 'yellow', 'red']
lst2 = lst1
lst1.append('purple')
print(lst1)
print(lst2)
```

Out[14]:
```
['black', 'white ', 'yellow', 'red', 'purple']
['black', 'white ', 'yellow', 'red', 'purple']
```

This behaviour occurs because both variables are assigned to the same memory space. To avoid this behavior, you need to create a new list. You can do this using `list()` or by using the `copy()` method. With this, you assign the lists to different memory spaces.

`lst2 = list(lst1)` or `lst2 = lst1.copy()`

In [15]:

```
lst1 = ['black', 'white ', 'yellow', 'red']
lst2 = list(lst1)
lst1.append('purple')
print(lst1)
print(lst2)
```

Out[15]:

```
['black', 'white ', 'yellow', 'red', 'purple']
['black', 'white ', 'yellow', 'red']
```

In [16]:

```
lst1 = ['black', 'white ', 'yellow', 'red']
lst2 = lst1.copy()
lst1.append('purple')
print(lst1)
print(lst2)
```

Out[16]:

```
['black', 'white ', 'yellow', 'red', 'purple']
['black', 'white ', 'yellow', 'red']
```

Sort Lists

Python has `sort()` function that will sort the elements in a list in ascending order, by default.

In [17]:

```
lst1 = ['black', 'white ', 'yellow', 'red', 'purple']
#Sort the list alphabetically
lst1.sort()
print(lst1)
```

Out[17]:

```
['black', 'purple', 'red', 'white ', 'yellow']
```

In [18]:

```
lst2 = listNumbers = [120, 50, 3, 200, 8.9, 3000, 90, 6.66 ]
#Sort the list numerically
lst2.sort()
print(lst2)
```

Out[18]:

```
[3, 6.66, 8.9, 50, 90, 120, 200, 3000]
```

If you want to sort the list in descending order, you will have to use `reverse = True`, as shown in the examples below.

In [19]:

```
lst1 = ['black', 'white ', 'yellow', 'red', 'purple']
#Sort the list descending
lst1.sort(reverse = True)
print(lst1)
```

Out[19]:

```
['yellow', 'white ', 'red', 'purple', 'black']
```

In [20]:
```
lst2 = listNumbers = [120, 50, 3, 200, 8.9, 3000, 90, 6.66 ]
#Sort the list descending
lst2.sort(reverse = True)
print(lst2)
```

Out[20]:
```
[3000, 200, 120, 90, 50, 8.9, 6.66, 3]
```

The `sort()` function is case-sensitive; thus, if you have a list with both capital and lower-case letters, it will sort all the capital letter elements first, before the lower-case letters.

In [21]:
```
lst1 = ['black', 'White ', 'Yellow', 'red', 'purple']
lst1.sort()
print(lst1)
```

Out[21]:
```
['White ', 'Yellow', 'black', 'purple', 'red']
```

In Python, we are allowed to customise your sort by using `key`. So, if you want to make your `sort()` function case-insensitive, you can do it as follows.

In [22]:
```
lst1 = ['black', 'White ', 'Yellow', 'red', 'purple']
lst1.sort(key = str.lower)
print(lst1)
```

Out[22]:
```
['black', 'purple', 'red', 'White ', 'Yellow']
```

Count List Elements

You can use the `count(value)` function to return the number of elements with the specified value in a given list. Note that the `value` that we search for could be any Python data type (e.g., *string*, *number* or even *list*).

In [23]:
```
lst1 = ['John', 91, [90, [False, 'Smith'], 19, True, 'David'], 'Denise', 91, 'David', 'Dav
id', 91]
print(lst1.count('john'))
print(lst1.count('John'))
print(lst1.count('David'))
print(lst1.count(91))
print(lst1.count(19))
print(lst1.count([90, [False, 'Smith'], 19, True, 'David']))
```

Out[23]:
```
0
1
2
3
0
1
```

Reverse List Elements

You can use the `reverse` function to reverse the order of the elements in a given list, as shown in the example below.

In [24]:
```
lst1 = ['black', 'White ', 'Yellow', 'red', 'purple']
lst1.reverse()
print(lst1)
```

Out[24]:
```
['purple', 'red', 'Yellow', 'White ', 'black']
```

Other List Operations

If you want to know which methods a list has, type the word 'list' with a dot and then the tab key in a code cell. A menu will be displayed with all the methods. If you want to know more about a method, type the word 'help', followed by the method inside of parenthesis e.g., `help(list.remove)`.

In [25]:
```
help(list.remove)
```

Out[25]:
```
Help on method_descriptor:

remove(self, value, /)
    Remove first occurrence of value.

    Raises ValueError if the value is not present.
```

In [26]:
```
help(list.count)
```

Out[26]:
```
Help on method_descriptor:

count(self, value, /)
    Return number of occurrences of value.
```

Your turn for some practice

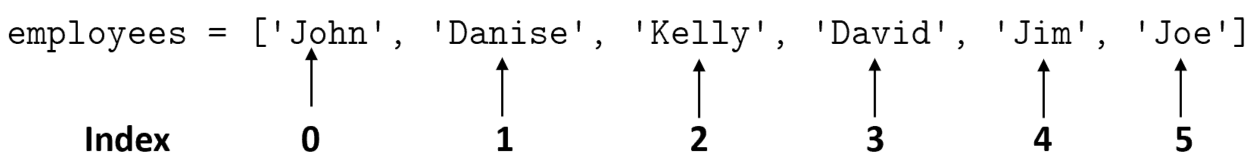
1. In your Jupyter Notebook, write Python code to create two lists and concatenate both lists in a third one. Create a fourth list with five copies of the first list (using the `*` operator), followed by two copies of the second list. Print the length of your third and fourth lists. Finally, use `in` and `not in` to print if an element is or is not in the fourth list.
2. In your Jupyter notebook, write code that uses each of the following list methods.
   - `clear`
   - `copy`
   - `count`
   - `extend`
   - `append`
   - `sort`
   - `pop`

- `reverse`

Add a conditional statement (if-else) to only call the method if it will not return an error. Add a markdown cell for each, explaining the purpose of each of the methods.

Accessing Elements in a List

To access the elements contained in a list via their position within the list, you can use square brackets. The number within the square brackets is called the **index**, as shown in the below image. The first element in a list is [0] and the last one is [n-1] where n is the number of elements in the list.



```
employees = ['John', 'Danise', 'Kelly', 'David', 'Jim', 'Joe']
```

```
employees[0] = 'John'
employees[1] = 'Danise'
employees[2] = 'Kelly'
employees[3] = 'David'
employees[4] = 'Jim'
employees[5] = 'Joe'
```

Let's have a look at more examples.

In [11]:
```
#access list element example
listNumbers = ['zero', 1, 2, 3, 'four', 5.0, 'six', 7, 8.0, 9]
listNumbers [0]
```

Out[11]:
```
'zero'
```

In [12]:
```
#access list element example
print(listNumbers[4],listNumbers[9])
```

Out[12]:
```
four 9
```

Your turn for some practice: List Index

1. What is the output of the following program?

```
my_list = ["John", [4, 8, 12, 16]]
print(my_list[0][1])
print(my_list[1][3])
```

2. Each list item has an index number. In the following list, which item has the index number of 1?

```
["John", "Harry", "Jesse", "John", "Harry", "Harry"]
```

Summary

In this section, you've covered a lot about lists. Lists allow you to store multiple data in a single variable. Key concepts we have covered:

- Lists with the same type of data
- Lists with different types of data
- Operators on lists (`+`, `*`, `in`, `not in`)
- List methods (e.g., `len()`, `append()`, `remove()`, `sort()`)
- List index (e.g., `mylist[0]`)

In this section, we did not cover how you could update list values since it involves *indexing*. We will look into this in a future topic of this module.

Readings

Read Sections 7.1 and 7.2.1, in:

Zhang, Y., (2015), ***An Introduction to Python and Computer Programming*** **(https://ap01.alma.exlibrisgroup.com/leganto/public/61ADELAIDE_INST/citation/2223796223590001811?auth=CAS)** ,(1st ed. Lecture Notes in Electrical Engineering 353), Springer, London.

You can also access all the readings from this course from the 'Course Readings' menu item on the left of your screen.

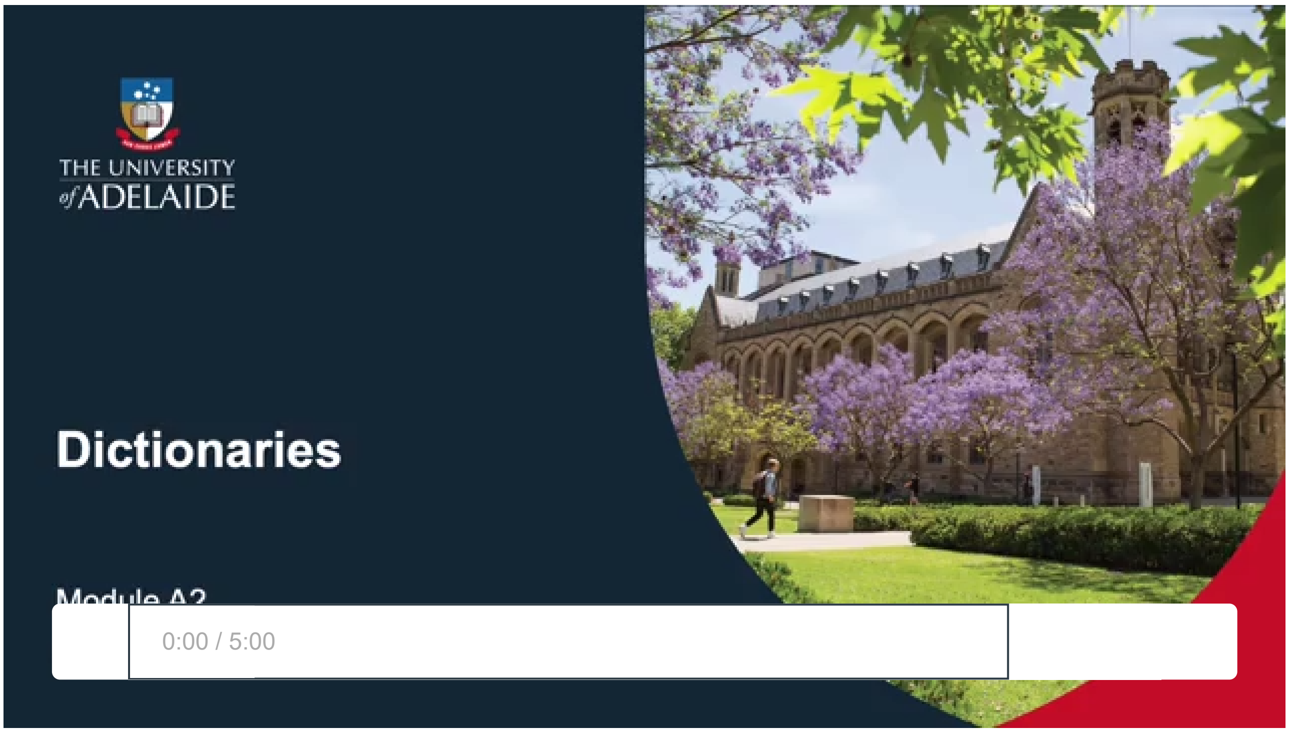
(h (h (h (h (h (h (h (h (h (h

A2 Topic 4: Dictionaries

Often you want to associate data values with a label rather than just having the data stored one after another in a list. For example, what if you wanted to store a group of people with their age? John 52, Siobhan 21, Ye 18... A list doesn't suit this very well. Fortunately, there is another structure that does: a dictionary.

A dictionary allows you to use the name as a 'key' for finding the desired value. For example, `age["John"]` will give the value 52.

The following video introduces what dictionaries are in Python.



Creating a Dictionary

To create a dictionary, you use curly brackets with *key:value* pairs separated by commas:

`age = {"John":52, "Siobhan":21, "Ye":18}`

You can also create dictionaries using a special construction method:

`age = dict(John=52, Siobhan=21, Ye=18)`

Note `=` is used instead of `:` when creating the dictionary with the `dict()` method!

You can use either approach to creating your list.

In [1]:
```
age1 = {"John":52, "Siobhan":21, "Ye":18}
print(type(age1))
```

```
age2 = dict(John=52, Siobhan=21, Ye=18)
print(type(age2))
print(len(age2))
```

Out[1]:

```
<class 'dict'>
<class 'dict'>
3
```

You aren't limited to strings. Your **key** can be any type. Well, almost. The type has to be *immutable*, which means it can't be modifiable. int, float, bool, string are all immutable, so your key can be any of these. Lists are an example of a mutable data type. You can change the values in a list without doing an assignment.

The type(s) of the **values** have no restrictions.

Note

You can also mix types for both the key and the values

`adelaideWeather = {'AvgTemp':24, 'AvgYearlyRainfall': 553, 'Climate': 'mediterranean'}`

You can even have your values be dictionaries!

`favouriteColours = {'John':{'blue', 'purple'}, 'Ye':{'red', 'yellow'}}`

The only restriction is that the key type(s) you choose must be immutable. The type(s) of the values have no restrictions.

It's time to practise creating dictionaries. Think of another example where you might want to store data as *key:value* pairs. In your Jupyter Notebook, create your own dictionaries using both ways of creating them.

Dictionaries do not allow duplicate keys. In other words, dictionaries cannot have two elements that have the same key. If duplicate keys were encountered, the values will be overwritten using existing values.

In [2]:

```
age = {"John":52, "Siobhan":21, "Ye":18, "John": 12}
print(age)
```

Out[2]:

```
{'John': 12, 'Siobhan': 21, 'Ye': 18}
```

In Python 3.6 and earlier, dictionaries are *unordered*, meaning the elements in the dictionary doesn't have a defined order; thus, you cannot refer to an element by using an index. However, from Python version 3.7 onwards, dictionaries are *ordered* (as lists), meaning the elements in a dictionary have a defined order, and that order won't change.

You can check the Python version from your Jupyter Notebook as follows.

```
import sys
print(sys.version)
```

Add Dictionary Elements

To add an element to a dictionary, you can use a new index key and assign a value to it, as shown in the example below.

In [3]:

```
details = {"fname":"John", "height":"190cm", "weight":"52kg", "age":30}
#add new dictionary elements
details["lname"] = "Smith"
details["interests"] = ["programming", "reading", "cooking", "gardening"]
print(details)
print(len(details))
```

Out[3]:

```
{'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 30, 'lname': 'Smith', 'inter
ests': ['programming', 'reading', 'cooking', 'gardening']}
6
```

Access Dictionary Elements

To access elements of a dictionary, you can use the specific key name inside of square brackets, as shown in the example below.

In [4]:

```
details = {'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 30, 'lname': 'Sm
ith', 'interests': ['programming', 'reading', 'cooking', 'gardening']}
#access dictionary elements
print(details["lname"])
print(details["fname"])
```

Out[4]:

```
Smith
John
```

Update Dictionary Elements

You can change the value of a specific element by using its key name. In addition, you can use the `update()` function with key:value pairs, as shown in the examples below.

In [5]:

```
details = {'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 30, 'lname': 'Sm
ith', 'interests': ['programming', 'reading', 'cooking', 'gardening']}
#update dictionary elements
details["lname"] = "Simon"
print(details)

details.update({"lname": "Denise"})
```

```
print(details)

details.update({"lname" : "David", "age": 12})
print(details)
```

Out[5]:

```
{'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 30, 'lname': 'Simon', 'inter
ests': ['programming', 'reading', 'cooking', 'gardening']}
{'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 30, 'lname': 'Denise', 'inte
rests': ['programming', 'reading', 'cooking', 'gardening']}
{'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 12, 'lname': 'David', 'inter
ests': ['programming', 'reading', 'cooking', 'gardening']}
```

Remove Dictionary Elements

To remove a dictionary element, use `pop()` function with the specified key name. Same as lists you can use `clear()` to empty the dictionary and `del` to delete the dictionary.

In [6]:

```
details = {'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 30, 'lname': 'Sm
ith', 'interests': ['programming', 'reading', 'cooking', 'gardening']}
#remove dictionary elements
details.pop("interests")
print(details)
details.pop("lname")
print(details)

#clear the dictionary
details.clear()
print(details)

#delete the dictionary
del details
print(details)
```

Out[6]:

```
{'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 30, 'lname': 'Smith'}
{'fname': 'John', 'height': '190cm', 'weight': '52kg', 'age': 30}
{}
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
ipykernel.py in <module>
     12 #delete the dictionary
     13 del details
---> 14 print(details)

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

Dictionary Operations

Now that you have learnt how to add, access, update and delete your data in a dictionary, let's use them with some other operations that you learnt with lists in the following example.

In [7]:

```
# Example working with Dictionaries

age = dict();  # create an empty dictionary

# add entries to the dictionary
```

```
age["Siobhan"] = 21
age["John"] = 52

# repr( ) returns the string representation of the dictionary
print("age dictionary: " + repr(age))

# len( ) gives the length of the dictionary
print("There are "+ str(len(age)) + " key:value pairs in the dictionary")

# copy( ) makes a copy of the dictionary
ageCopy = age.copy()
print("age dictionary copy: " + repr(ageCopy)) # return the string representation of the
dictionary

# difference between == and is
print("copies are equivalent? ", age == ageCopy) # == and != check if lists have same ke
y:value pairs
print("are age and ageCopy the same dictionary? ", age is ageCopy) # note the difference
between == and is!

# removing elements from the dictionary
# pop( ) removes and returns the value
ageJohn = age.pop("John")
print(ageJohn)

# del ( ) removes but doesn't return the return value
del age["Siobhan"]

# John and Siobhan have both been removed so list should now be empty
print("age dictionary: " + repr(age))

# but the copy of the dictionary hasn't been changed
print("age dictionary copy: " + repr(ageCopy))
print("copies are equivalent? ", age == ageCopy)

# clear( ) removes everything in a dictionary
ageCopy.clear()
print("age dictionary copy: " + repr(ageCopy)) # return the string representation of the
dictionary
```

Out[7]:

```
age dictionary: {'Siobhan': 21, 'John': 52}
There are 2 key:value pairs in the dictionary
age dictionary copy: {'Siobhan': 21, 'John': 52}
copies are equivalent?  True
are age and ageCopy the same dictionary?  False
52
age dictionary: {}
age dictionary copy: {'Siobhan': 21, 'John': 52}
copies are equivalent?  False
age dictionary copy: {}
```

Your turn for some practice

1. In your Jupyter Notebook, add four key:value pairs to your dictionary.
2. Check the number of key:value pairs in your dictionary using `len()`.
3. Change the value of one of your key:value pairs. Pop a key:value pair from your dictionary and print the returned value.
4. Finally, delete a key:value pair and then clear your dictionary.

Checking if a key is in your Dictionary

What happens if you try to delete a key that isn't in your dictionary?

To avoid an error trying to remove a key that doesn't exist in the dictionary, you should check first if the key exists.

In [8]:

```
age = { }
if "John" in age:
    print("John is in age dictionary")
    del age["John"]
```

Out[8]:

In your Jupyter notebook, run the code when the key "John" is in the age list, and when it isn't.

Manipulating Dictionary Data

Dictionaries, like lists, are useful as a container for your data sequence. As a sequence container, you can use iteration (loops) over all the items in your dictionary. For example, suppose a year has passed, and you want to update everyone's age. There are three dictionary functions that can help you do this:

- `items()` get all the key:value pairs
- `keys()` get all the keys
- `values()` get all the values

Here are some examples of how to use each.

In [9]:

```
age = {"John":52, "Siobhan":21, "Ye":18}

for k,v in age.items():
    print(k, ":", v)
print()

for k in age.keys():
    print(k)
print()

for v in age.values():
    print(v)
print()
```

Out[9]:

```
John : 52
Siobhan : 21
Ye : 18

John
Siobhan
```

```
Ye

52
21
18
```

Returning to the problem of updating everyone's age. You want to update each value for the available keys, the equivalent of:

`age["John"] = age["John"] + 1`

`age["Siobhan"] = age["Siobhan"] + 1`

`age["Ye"] = age["Ye"] + 1`

For this, you need each of the keys, so you should choose the `keys( )` function.

In [10]:

```python
age = {"John":52, "Siobhan":21, "Ye":18}

for k in age.keys():
    age[k] = age[k] + 1

print(repr(age))
```

Out[10]:

```
{'John': 53, 'Siobhan': 22, 'Ye': 19}
```

Nested Dictionaries

Same as lists, you can have dictionaries inside a dictionary, which is called nested dictionaries. See below for an example of a dictionary that contains three dictionaries.

In [11]:

```python
from pprint import pprint
students = {'student1' : {'name':'John', 'age':19, 'interests':['programming', 'readin
g', 'cooking', 'gardening']}, 'student2' : {'name':'Danise', 'age':12, 'interests':['pro
gramming']}, 'student3' : {'name':'Simon', 'age':9, 'interests':['watching TV']}}
pprint(students)
print(len(students))
```

Out[11]:

```
{'student1': {'age': 19,
              'interests': ['programming', 'reading', 'cooking', 'gardening'],
              'name': 'John'},
 'student2': {'age': 12, 'interests': ['programming'], 'name': 'Danise'},
 'student3': {'age': 9, 'interests': ['watching TV'], 'name': 'Simon'}}
3
```

If you want to add three dictionaries into a dictionary, you can also do it as follows.

In [12]:

```python
from pprint import pprint
student1 = {'name':'John', 'age':19, 'interests':['programming', 'reading', 'cooking',
'gardening']}
student2 = {'name':'Danise', 'age':12, 'interests':['programming']}
student3 = {'name':'Simon', 'age':9, 'interests':['watching TV']}
students = {'student1': student1, 'student2': student2, 'student3': student3}
pprint(students)
print(len(students))
```

Out[12]:
```
{'student1': {'age': 19,
              'interests': ['programming', 'reading', 'cooking', 'gardening'],
              'name': 'John'},
 'student2': {'age': 12, 'interests': ['programming'], 'name': 'Danise'},
 'student3': {'age': 9, 'interests': ['watching TV'], 'name': 'Simon'}}
3
```

Summary

You've seen that dictionaries are lists with labels. Key concepts you have covered in this section include:

- creating dictionaries
- adding, accessing, updating and removing values of dictionary elements
- other operations such as using copy() to copy dictionaries
- useful functions for looping over and modifying the values in a list: `items()`, `keys()` and `values()`

Readings

Read Sections 8.2, 8.2.1 and 8.2.2, in:

> Zhang, Y., (2015), ***An Introduction to Python and Computer Programming*** **(https://adelaide.leganto.exlibrisgroup.com/leganto/public/61ADELAIDE_INST/citation/2223796223590001811?auth=SAML)** ,(1st ed. Lecture Notes in Electrical Engineering 353), Springer, London.

You can also access all the readings from this course from the 'Course Readings' menu item on the left of your screen.

(h (h (h (h (h (h (h (h (h (h

A2 Topic 6: More on Strings

Accessing the characters in a string

Every character in a string is stored at a specific string *index*. What is an index in a string?

An index is the position of the character in the string. In Python, each of a string's characters corresponds to an index number, starting with the index number 0. See the figure below.

string	H	E	L	L	O		W	O	R	L	D	!
index	0	1	2	3	4	5	6	7	8	9	10	11

In [1]:
```
#Getting the letter of a specific position or index
str1 = 'HELLO WORLD!'
#To access a specific character in a string write the string variable name followed by squ
are brackets with the position number you want.
print(str1[6])
print(str1[9])
```

Out[1]:
```
W
L
```

What happens if we try to access some index that does not exist in the given string?

In [2]:
```
str1 = 'HELLO WORLD!'

print(str1[6])
print(str1[19])
```

Out[2]:
```
W
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
ipykernel_1.py in <module>
      2 #To access a specific character in a string write the string variable name followe
d by square brackets with the position number you want.
      3 print(str1[6])
----> 4 print(str1[19])

IndexError: string index out of range
```

We get an `IndexError` since we ask for something that doesn't exist in the given string. In Python, a string is a single-dimensional array of characters. Indexes in Python programming start at 0. This means that the maximum index for any string will always be length-1. Therefore, you will get an `IndexError`, if you try to access an index that is bigger than the length.

Finding the first occurrence of a specified value in a string

We can also do the reverse; finding the first index, where a specific character or a substring is located in the string using `index()`. Note that `index()` performs a case-sensitive search.

In [3]:
```
# Performing index(x)
str1 = 'HELLO WORLD!'
print('Position of the character !: ', str1.index('!'))
print('Position of the substring LLO: ', str1.index('LLO'))
print('Position of the blank space: ', str1.index(' '))
```

Out[3]:
```
Position of the character !:  11
Position of the character L:  2
Position of the blank space:  5
```

Now, let's try to find the first occurrence of a substring that does not exist.

In [4]:
```
str1 = 'HELLO WORLD!'
print('Position of the character !: ', str1.index('!'))
print('Position of the substring LLO: ', str1.index('LLO'))
print('Position of the substring #!: ', str1.index('#!'))
```

Out[4]:
```
Position of the character !:  11
Position of the substring LLO:  2
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
ipykernel_1.py in <module>
      3 print('Position of the character !: ', str1.index('!'))
      4 print('Position of the substring LLO: ', str1.index('LLO'))
----> 5 print('Position of the substring #!: ', str1.index('#!'))

ValueError: substring not found
```

In `index()`, you can also define from where the search should start and end as follows. If you did not define the start and end, it will search in the entire string as you saw above.

```
string.index(value, start, end)
```

In [5]:
```
str1 = 'HELLO WORLD!'
print('Position of the substring LLO: ', str1.index('LLO', 2, 6))
```

Out[5]:
```
Position of the substring LLO:  2
```

What happens if `index()` could not find a substring that we specified? It throws `ValueError`, if the specified substring is not found in a given string, as shown in the example below.

In [6]:
```
str1 = 'HELLO WORLD!'
print('Position of the substring lLO: ', str1.index('lLO'))
```

Out[6]:
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
ipykernel_1.py in <module>
      1 str1 = 'HELLO WORLD!'
----> 2 print('Position of the substring lLO: ', str1.index('lLO'))

ValueError: substring not found
```

In addition to `index()`, you can also use `find()` that perform the same operation (which is identifying the index of the first occurrence of a substring in a given string). The only difference between `index()` and `find()` is that `find()` returns -1, if a substring is not found.

In [7]:
```
str1 = 'HELLO WORLD!'
print('Position of the substring lLO: ', str1.find('lLO'))
print('Position of the substring LLO: ', str1.find('LLO'))
```

Out[7]:
```
Position of the substring lLO:  -1
Position of the substring LLO:  2
```

Slicing a String

A *slice* is used to pick out part of a string. It usually takes three values:

1. start — starting index where the slicing of the string starts.
2. stop — index until which the slicing takes place. The slicing stops at the length of the string.
3. step — integer value which determines the increment between each index for slicing.

Example: `str1[start:stop:step]`

You don't have to provide all of the parameters; just the ones you want to specify. If you leave out the step parameter, it will step by 1. If you leave out start, it will start from the beginning of the string. If you leave out stop, it will go to the end of the string.

You can also use negative values. For the start and stop, it will be the distance of the index from the end of the string, instead of from the beginning. For step, a negative number will cause it to go from the end characters towards the start.

In [8]:
```
# Examples with slices
str1 = 'HELLO WORLD!' # This string has 12 characters
print('length of str1->',len(str1))
print('str1[0:5]->',str1[0:5])#without step
print('str1[0:5:1]->',str1[0:5:1])#with step
print('str1[ :5]->',str1[:5])#When the start position is not specified for default it star
ts in 0
print('str1[6:12]->',str1[6:12])#Return the characters from position 6 to 12
print('str1[6: ]->',str1[6: ])#When the stop position is not specified for default it stop
s at the end of the string
print('str1[ : :2]->',str1[ : :2])#Return characters every 2 positions
print('str1[-3: ]->',str1[-3: ])#Return the last 3 characters of the string
print('str1[-6:-1]->',str1[-6:-1])#Use negative indexes to start slicing from the end of t
he string
print('str1[0:6:2]->',str1[ 0:6:2])#Return the characters from position 0 to 6 every 2 pos
itions
print('str1[ : :-1]->',str1[ : :-1])#a negative step reverses the string
print('str1[:]->',str1[:])#Return all the characters of the string
```

Out[8]:
```
length of str1-> 12
str1[0:5]-> HELLO
str1[0:5:1]-> HELLO
str1[ :5]-> HELLO
str1[6:12]-> WORLD!
str1[6: ]-> WORLD!
str1[ : :2]-> HLOWRD
str1[-3: ]-> LD!
str1[-6:-1]-> WORLD
str1[0:6:2]-> HLO
```

```
str1[ : :-1]-> !DLROW OLLEH
str1[:]-> HELLO WORLD!
```

Your turn for some practice: Slicing a String

1. In your Jupyter Notebook, write the code that will take the variable *alphabet* and slice it to get the desired outputs.

   `alphabet ='abcdefghijklmnopqrstuvwxyz'`

   These are the desired outputs:

   1. abcde
   2. fghijklmno
   3. abcdefghijklmnopqrstu
   4. acegikmoqsuwy
   5. aeimquy
   6. zyxwvutsrqponmlkjihgfedcba

Splitting a String into a List

In Python, we use the `split()` function to split a given string into a list.

```
str.split(separator, maxsplit)
```

- `separator` *(optional)*: You can specify the separator that you want to use when splitting the string. The default separator is any whitespace.
- `maxsplit` *(optional)*: This defines how many splits you want. The outputted list will contain the specified number of items *plus one*. The default will be -1, indicating all occurrences.

Let's look at a few examples to understand this.

In [9]:
```
#Example of split separators
txt = "hello, my name is john, \t i am an student from the university of adelaide, \n i lo
ve python progamming"
print("Default separator: ", txt.split()) #The default separator will be any whitespace
print("\", \" separator: ", txt.split(", ")) #Using comma, followed by a space as a separa
tor
print("\"\\n\" separator: ", txt.split("\n")) #Using new line
```

Out[9]:
```
Default separator:  ['hello,', 'my', 'name', 'is', 'john,', 'i', 'am', 'an', 'student', 'f
rom', 'the', 'university', 'of', 'adelaide,', 'i', 'love', 'python', 'progamming']
", " separator:  ['hello', 'my name is john', '\t i am an student from the university of a
delaide', '\n i love python progamming']
"\n" separator:  ['hello, my name is john, \t i am an student from the university of adela
ide, ', ' i love python progamming']
```

In [10]:
```
#Example of split separators and maxsplit
txt = "hello, john, university of adelaide, python progamming, jupyter notebook, happy cod
ing"
```

```
#without maxsplit
lst1 = txt.split(", ")
print(lst1)
print(len(lst1))
#with maxsplit
lst2 = txt.split(", ", 2)
print(lst2)
print(len(lst2))
```

Out[10]:

```
['hello', 'john', 'university of adelaide', 'python progamming', 'jupyter notebook', 'happ
y coding']
6
['hello', 'john', 'university of adelaide, python progamming, jupyter notebook, happy codi
ng']
3
```

Joining an iterable into one String

In Python, you can use `join()` to take all items in an iterable and join them into one string. The iterable could be a list, a dictionary (that you already know), or a tuple, a set (that you will learn in Week 3 of this module).
Let's look at an example to see how we can join all items in a list into a string.

In [11]:

```
list1 = ['hello', 'john', 'university of adelaide', 'python progamming', 'jupyter noteboo
k', 'happy coding']

print("#".join(list1)) #using a hash character as separator
print(" ".join(list1))#using a whitespace as separator
```

Out[11]:

```
hello#john#university of adelaide#python progamming#jupyter notebook#happy coding
hello john university of adelaide python progamming jupyter notebook happy coding
```

Your turn for some practice: Split and Join

1. Given the string "I love programming", split the string on a `" "` (space) delimiter and join using a `-` hyphen.
2. What is the output of the following code?

```
def split_string(string):
    list_string = string.split(' ')

    return list_string

def join_string(list_string):
    string = ' !'.join(list_string)

    return string

string = 'My name is John'
list_string = split_string(string)
print(list_string)
new_string = join_string(list_string)
print(new_string)
```

A2 | Topic 6: More on Lists

A2 Topic 3: More on Lists

Accessing Elements in a List

To access the elements contained in a list via their position within the list, you can use square brackets. The number within the square brackets is called the **index**, as you saw previously with strings. The first element in a list is [0] and the last one is [n-1] where n is the number of elements in the list.

In [1]:
```
#access list element example
listNumbers = ['zero', 1, 2, 3, 'four', 5.0, 'six', 7, 8.0, 9]
listNumbers [0]
```

Out[1]:
```
'zero'
```

In [2]:
```
#access list element example
print(listNumbers[4],listNumbers[9])
```

Out[2]:
```
four 9
```

Slicing a List

Slicing gets a part of the list. The slice syntax is `[start:stop:step]` and returns a subsequence of the list as a new list:

- **start** index is optional and when it is omitted, the default value is 0.
- **stop** index is optional and the default value is the length of the list.
- **step** is the space between the values. This value is optional and the default value is 1. A negative step reverses the direction of the iteration.

Does this remind you of the *string slicing* and `range()` function? The `[start:stop:step]` format is common when you're working with indexes of a sequence in Python.

In [3]:
```
#slicing example
listNumbers = ['zero', 1, 2, 3, 'four', 5.0, 'six', 7, 8.0, 9]
# get items starting at 0 and stopping at, not including, 3
listNumbers[0:3]
```

Out[3]:
```
['zero', 1, 2]
```

In [4]:
```
print(listNumbers[:3]) #Omit start by default starts in the position 0
```

Out[4]:
```
['zero', 1, 2]
```

In [5]:
```
listNumbers[3:] #Omit stop by default goes until the length of the list
```

Out[5]:
```
[3, 'four', 5.0, 'six', 7, 8.0, 9]
```

In [6]:
```
length = len(listNumbers)
print('length of the list is',length)
print(listNumbers[3:length])
```

Out[6]:
```
length of the list is 10
[3, 'four', 5.0, 'six', 7, 8.0, 9]
```

In [7]:
```
# using a negative value
# recall that negative values count from the end
print('listNumbers[-1] = ',listNumbers[-1])
print('listNumbers[-2] = ',listNumbers[-2])
```

Out[7]:
```
listNumbers[-1] =  9
listNumbers[-2] =  8.0
```

In [8]:
```
#using a negative value with start
listNumbers[0:-1]#get all the elements except the last one
```

Out[8]:
```
['zero', 1, 2, 3, 'four', 5.0, 'six', 7, 8.0]
```

In [9]:
```
#using a negative value in start
listNumbers[-3:] #get the last three elements
```

Out[9]:
```
[7, 8.0, 9]
```

In [10]:
```
#using a negative value -1 for the step
listNumbers[::-1] #reverse the list

# note that when the step is negative, if the start:stop aren't specified,
# the start becomes the end by default and the stop becomes the beginning
# this is equivalent to
listNumbers[10:0:-1]
```

Out[10]:
```
[9, 8.0, 7, 'six', 5.0, 'four', 3, 2, 1]
```

In [11]:
```
#Slicing with two steps
listNumbers[0:10:2]
```

Out[11]:
```
['zero', 2, 'four', 'six', 8.0]
```

Info

- The first element in a list has an index 0.
- When specifying a range of indexes, the return value will be a new list with the specified elements (e.g., `lst1[2:6]`)
- When using `lst1[2:6]`, the search will start at index 2 (included) of `lst1` and end at index 6 (not included).
- Specify negative indexes if you want to start the search from the end of a given list.
- When using `lst1[-6:-2]`, the search returns the elements from -6 to including -2.

Your turn for some practice

1. In your Jupyter Notebook, write the code that will take the list and slice it to extract every third element from the end to the beginning of the list (*Hint:* Think how you will use `step` to solve this problem).
2. Given the list `lst1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]`, obtain the following outputs using list index and slicing.

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

Setting Value in a List

Setting the value in a list takes the form of the assignment statement e.g. `list1[index] = value`.

Also, you can use slice to assign a value, or values, in a specific slice of the list.

In [12]:

```
#Example of setting the value in a specific list position
list1 = [1, 2, 3, 'Hello', 5, 'six']
print('list1 before assignment  ',list1)
list1[3] = 'four'
print('list1 after assignment   ',list1)
```

Out[12]:

```
list1 before assignment   [1, 2, 3, 'Hello', 5, 'six']
list1 after assignment    [1, 2, 3, 'four', 5, 'six']
```

In [13]:

```
#Example of setting the values in a slice of the list
list1 = [1.4, 1, 20.1, 5, 19.3, 91, 30, 10.2, 9, 26]
```

```
list1[2:4] = [9, 12]
print(list1)
```

Out[13]:

```
[1.4, 1, 9, 12, 19.3, 91, 30, 10.2, 9, 26]
```

If you insert *more* elements than you replace, the new elements will be inserted where you specified, and the remaining elements will move accordingly. However, the length of the list will change when the number of elements inserted doesn't match the number of elements replaced.

In [14]:

```
#Example of setting more number of values in a slice of the list
list1 = [1.4, 1, 20.1, 5, 19.3, 91, 30, 10.2, 9, 26]
print('previous length: ', len(list1))
list1[2:4] = [9, 12, 19]
print(list1)
print('new length: ', len(list1))
```

Out[14]:

```
previous length:  10
[1.4, 1, 9, 12, 19, 19.3, 91, 30, 10.2, 9, 26]
new length:  11
```

Similarly, if you insert *fewer* elements than you replace, the new elements will be inserted where you specified, and the remaining elements will move accordingly. Once again, the length of the list will change.

In [15]:

```
#Example of setting less number of values in a slice of the list
list1 = [1.4, 1, 20.1, 5, 19.3, 91, 30, 10.2, 9, 26]
print('previous length: ', len(list1))
list1[2:4] = [9]
print(list1)
print('new length: ', len(list1))
```

Out[15]:

```
previous length:  10
[1.4, 1, 9, 19.3, 91, 30, 10.2, 9, 26]
new length:  9
```

Insert List Elements

In the previous module, you learned that you can add an element to the end of the list, using `append()`. To insert an element at a specific index you have to use `insert()` function. More specifically, the `insert(index, element)` function is used to add an element in a specific position indicated by index.

In [16]:

```
#Example insert an element
colors = ['black', 'white', 'yellow', 'red']
print('colors before insert ', colors)
colors.insert(3,'green')
print('colors after insert ', colors)
```

Out[16]:

```
colors before insert  ['black', 'white', 'yellow', 'red']
colors after insert  ['black', 'white', 'yellow', 'green', 'red']
```

Remove Specified Index

In the previous module, you learnt `remove()`, `pop()`, `del` and `clear`. Please revisit the previous content you learned, if you do not remember their intended functionalities.

Previously, you saw that the `pop()` function removed the last element in the list. Now, we will learn how to use the `pop()` function to remove the specified index. In the below example, you can see that we have removed the second element in the list by specifying its index value (which is 1).

In [17]:
```
#Example remove specified index
colors = ['black', 'white', 'yellow', 'red']
colors.pop(1)
print(colors)
```

Out[17]:
```
['black', 'yellow', 'red']
```

Why Aren't Strings 'Mutable' When Lists Are?

This is a subtle difference, but the key to understanding this is to think about whether you can change the data itself, or whether you have to assign the result.

A list has several methods that allow you to change it: `append( )`, `insert( )`, `sort( )`... strings have none of these. You can't change a string once it's created. Of course, you can reassign the variable to a different string:

`myString = 'John'`

`myString = myString + ' Ye'`

But notice here you haven't actually changed the original string 'John'. Instead, you have changed `myString` to refer to a *new* string: 'John Ye'.

In contrast, in:

`nameList = ['John']`

`nameList.append('Ye')`

You have actually changed the original list that just contained 'John' to now also have 'Ye'. Note that you didn't have to reassign `nameList` to refer to a new list. You didn't have to have:

`nameList = nameList.append('Ye')`

If you can change the type directly through a method, it is mutable. If you have to use an assignment (because there are no methods provided by the type that allow you to change it), it is immutable.

A2 Topic 6: List Comprehension

List Comprehension

If you want to separate the letters of the word "Hello!" and add these letters to a new list, you will probably come up with a solution as follows using for loops.

In [1]:
```
new_lst = []

for letter in 'Hello!':
    new_lst.append(letter)

print(new_lst)
```

Out[1]:
```
['H', 'e', 'l', 'l', 'o', '!']
```

In Python, we can do this in a lot easier way using *List Comprehension*. More specifically, list comprehension offers a shorter syntax to define and create a new list based on the values in the existing lists. Let's look at how we can perform the above program using list comprehension.

In [2]:
```
new_lst = [letter for letter in 'Hello!']
print(new_lst)
```

Out[2]:
```
['H', 'e', 'l', 'l', 'o', '!']
```

The basic syntax of list compresion is:

```
[expression for item in iterable]
```

which returns a new list, leaving the old list unchanged.

Try out the following list comprehensions by yourself:

```
times_one = [0, 1, 1, 2, 3, 5, 8]
times_two = [num * 2 for num in times_one]
times_two
```

```
[0, 2, 2, 4, 6, 10, 16]
```

and:

```
quarks = ["Up", "Down", "Charm", "Strange", "Top", "Botttom"]
lower_first_letter = [quark[0].lower() for quark in quarks]
lower_first_letter
```

```
['u', 'd', 'c', 's', 't', 'b']
```

The manipulations of lists using comphrensions are concise and Pythonic and may be used to create lists and dictionaries from existing data.

Conditionals in List Comprehension

What if you wanted to construct a new list `even_numbers` and add all the odd numbers in the range from 0-9 into this new list. You will probably come up with the follwoing Python code to do this.

In [3]:
```
lst1 = []

for i in range(10):
    if i%2 == 0:
        lst1.append(i)
print(lst1)
```

Out[3]:
```
[0, 2, 4, 6, 8]
```

Luckily, Python also allows you to construct list comprehensions with conditional statements to have a shorter syntax to perform such operations. In this example, we use a single `if` statement. The, syntax of a list comprehension using one `if` statement looks as follows:

```
[expression for item in iterable if condition == True]
```

In the following example, we construct a new list `even_numbers` and add all the odd numbers in the range from 0-9 to this new list.

In [3]:
```
even_numbers = [x for x in range(10) if x%2 == 0]
print(even_numbers)
```

Out[3]:
```
[0, 2, 4, 6, 8]
```

Now let's try to create a new list that contains numbers that are divisible by both 2 and 5 in the range from 0-99.

In [4]:
```
num_lst = [y for y in range(100) if (y%2 == 0 and y%5 == 0)]
print(num_lst)
```

Out[4]:
```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

What if you want to perform `if-else` statements. For example, Let's look at the following code that checks the numbers from 0-9 and determine if its odd or even.

In [5]:
```
lst1 = []

for i in range(10):
    if i%2 == 0:
        lst1.append("even")
    else:
        lst1.append("odd")
print(lst1)
```

Out[5]:
```
['even', 'odd', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'even', 'odd']
```

You can do the same operations using a list comprehension as follows.

In [6]:
```
lst1 = ["even" if i%2 == 0 else "odd" for i in range(10)]
print(lst1)
```

Out[6]:
```
['even', 'odd', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'even', 'odd']
```

Nested Loops in List Comprehension

You can also perform nested loops in list comprehension.

Suppose you wanted to perform transpose of a matrix. The transpose of a matrix is an operator which flips a matrix over its diagonal. An example of a transposed matrix is given below.

$$\begin{bmatrix} 1 & 9 \\ 2 & 0 \\ 9 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 & 9 \\ 9 & 0 & 6 \end{bmatrix}$$

This requires the use of nested for loop. Forst we will see how to do this using normal for loops.

In [7]:
```
transposed_matrix = []
matrix = [[1, 9], [2, 0], [9, 6]]

for i in range(2):
    transposed_row = []

    for row in matrix:
        transposed_row.append(row[i])
    transposed_matrix.append(transposed_row)

print(transposed_matrix)
```

Out[7]:
```
[[1, 2, 9], [9, 0, 6]]
```

Now, let's see how to do this using a list comprehension.

In [8]:
```
matrix = [[1, 9], [2, 0], [9, 6]]

transposed_matrix = [[row[i] for row in matrix] for i in range(2)]
print(transposed_matrix)
```

Out[8]:
```
[[1, 2, 9], [9, 0, 6]]
```

Note that the nested loops in the list comprehension do not work like normal nested loops. In the above program, `for i in range(2)` is executed before `row[i] for row in matrix`. Therefore, a value is assigned to `i` at first, and then item directed by `row[i]` is appended into the `transposed_matrix` variable.

Important Tip

- List comprehension allows you to create compact code when you want to create a new list based on the values of an existing list. However, do not write very long list comprehensions in one line to ensure that your code is readable, and easy to debug.
- Even though all list comprehensions can be rewritten using for loops, you can’t rewrite every for loops in the form of list comprehensions.

A2 | Topic 6: Tuples & Sets

A2 Topic 6: Tuples & Sets

You have come a long way. Now you have a very good understanding of the two non-primitive data types; lists and dictionaries. You learnt basics on lists in Topic 4 and expanded that knowledge in Topic 5. You also learnt about dictionaries in Topic 4.

In this topic, we will introduce you to tuples and sets.

Tuples

You may have already seen the tuple data type in the previous topics of this section. For example, when returning multiple values from functions you saw tuples.

In [1]:

```
def perform_calculations(num1, num2):
    return num1+num2, num1-num2, num1*num2, num1/num2

print(perform_calculations(20,2))
```

Out[1]:

```
(22, 18, 40, 10.0)
```

You also saw tuples when working with arbitrary arguments, as shown in the example below.

In [2]:

```
def construct_login_greeting(*args):
    print(args)
    print(type(args))

construct_login_greeting("David", "John", "Smith", "Denise")
```

Out[2]:

```
('David', 'John', 'Smith', 'Denise')
<class 'tuple'>
```

Let's have a look into more details in tuples. In Python, tuples are written with round brackets. Tuple elements are ordered, indexed, unchangeable, and allow duplicate values. Tuple elements can be of any data type. It can contain different data types too. Let's look at an example of a tuple.

In [3]:

```
details = ("John", "Smith", 30, "190cm", True, "male", True, 30, "male")
print(type(details))
print(len(details))
```

Out[3]:

```
<class 'tuple'>
9
```

If you want to have a tuple with only one element, you will have to add a comma after the element. Otherwise, Python won't recognise it as a tuple.

In [4]:
```
details = ("John")
print(type(details))

details = ("John",)
print(type(details))
```

Out[4]:
```
<class 'str'>
<class 'tuple'>
```

Let's do some operations with tuples. You will see that you have learnt these operations previously.

In [5]:
```
details = ("John", "Smith", 30, "190cm", True, "male", True, 30, "male")
#access tuple elements
print(details[0])
print(details[:-1])
print(details[1:2])
print(details[::2])

#multiply tuples
print(details*2)

#join tuples
details1 = ("Smith", 19)
print(details+details1)

#while loops
print("Starting while loop:")
i = 0
while i < len(details):
    if "m" in str(details[i]):
        print(details[i])
    i += 1
print()

#for loops
print("Starting for loop:")
for ele in details[:3]:
    print(ele)
print()

#there are two built-in functions that you can use on tuples
#index() function
print(details.index("male"))

#count() function
print(details.count("male"))
```

Out[5]:
```
John
('John', 'Smith', 30, '190cm', True, 'male', True, 30)
('Smith',)
('John', 30, True, True, 'male')
('John', 'Smith', 30, '190cm', True, 'male', True, 30, 'male', 'John', 'Smith', 30, '190c
m', True, 'male', True, 30, 'male')
('John', 'Smith', 30, '190cm', True, 'male', True, 30, 'male', 'Smith', 19)
Starting while loop:
Smith
190cm
male
male

Starting for loop:
```

```
John
Smith
30

5
2
```

Your turn for some practice: Tuples

1. Write a Python program to find the Maximum and Minimum K elements in Tuple

Example 1

```
The original tuple is : (5, 20, 3, 7, 6, 8)
k value: 2
The extracted values : (3, 5, 8, 20)
```

Example 2

```
The original tuple is : (5, 20, 3, 7, 6, 8)
k value: 1
The extracted values : (3, 20)
```

2. Write a Python program to create a list of tuples from a given list having a number and its cube in each tuple.

```
Input: list = [1, 2, 3]
Output: [(1, 1), (2, 8), (3, 27)]

Input: list = [9, 5, 6]
Output: [(9, 729), (5, 125), (6, 216)]
```

3. Write a Python program to find the sum of tuple elements.

```
The original tuple is : (7, 8, 9, 1, 10, 7)
The summation of tuple elements are: 42
```

4. Write a Python program to get the modulus tuple from two given tuples.

```
The original tuple 1 : (10, 4, 5, 6)
The original tuple 2 : (5, 6, 7, 5)
The modulus tuple : (0, 4, 5, 1)
```

Sets

In Python, sets are written with curly brackets. They are unordered, unchangeable (but you can remove elements and add new elements), unindexed and do not allow duplicate values. Let's look at an example of a set.

In [6]:
```
details = {"John", "Smith", 30, "190cm", True, "male", True, 30, "male"}
print(type(details))
print(details) #duplicates have been removed and the order has changed
print(len(details))
```

Out[6]:
```
<class 'set'>
{True, '190cm', 'male', 'Smith', 'John', 30}
6
```

Once you have created a set, you cannot change its elements, but you can add new elements or remove elements, as shown in the example below.

In [7]:
```
details = {True, '190cm', 'male', 'Smith', 'John', 30}
#adding new elements
details.add(12)
print(details)
#removing elements
details.remove(30)
print(details)
```

Out[7]:
```
{True, '190cm', 'male', 'Smith', 'John', 12, 30}
{True, '190cm', 'male', 'Smith', 'John', 12}
```

Now, let's perform some basic set operations such as `union`, `intersection`, `difference`, `isdisjoint()` and `issubset()`.

In [8]:
```
x = {"John", "Smith", "David", "Danise", "Simon", "Jane"}
y = {"James", "John", "Jane", "Miles"}
print("Union:", x.union(y)) #union() returns a set that contains all elements from both se
ts (duplicates are excluded)
print("Intersection:", x.intersection(y)) #intersection() returns a set that contains the
elements that exist in both sets
print("Difference:", x.difference(y)) #difference() returns a set that contains the elemen
ts that only exist in x, and not in y
print("Is Disjoint?", x.isdisjoint(y)) #isdisjoint() returns True if none of the elements
are present in both sets, otherwise it returns False
print("Is Subset?", x.issubset(y)) #issubset() returns True if all elements in the set exi
sts in the specified set, otherwise it retuns False
```

Out[8]:
```
Union: {'James', 'Danise', 'David', 'Smith', 'Simon', 'Jane', 'Miles', 'John'}
Intersection: {'Jane', 'John'}
Difference: {'David', 'Smith', 'Simon', 'Danise'}
Is Disjoint? False
Is Subset? False
```

Now, try swapping the position of `x` and `y` sets in the set operations to see whether the results get changed. If changed, observe why it gets changed by inspecting the set elements.

Your turn for some practice: Sets

1. Write a Python program to iterate over the elements in the following set.

```
test_set = set("python")
```

2. Write a Python program to find common elements in three lists using sets.

```
Input : ar1 = [1, 5, 10, 20, 40, 80]
        ar2 = [6, 7, 20, 80, 100]
        ar3 = [3, 4, 15, 20, 30, 70, 80, 120]

Output : [80, 20]

Input : ar1 = [1, 5, 5]
        ar2 = [3, 4, 5, 5, 10]
        ar3 = [5, 5, 10, 20]

Output : [5]
```

3. Write a Python program to find the difference between two lists.

```
Input :
list1 = [10, 15, 20, 25, 30, 35, 40]
list2 = [25, 40, 35]
Output :
[10, 20, 30, 15]
```

Quick Check

Which of the following are true for objects of Python’s set type?

The order of elements in a set is significant.

A given element can’t appear in a set more than once.

Check Answer

Quick Comparison

In this section, we will compare the properties of the 4 built-in data types in Python used to store collections of data, which are lists, dictionaries, tuples and sets. If you are unsure or unclear of any of the properties mentioned below, please revisit the course content.

Lists	Dictionaries	Tuples	Sets
Allow duplicate elements	No duplicate elements	Allow duplicate elements	No duplicate elements
Changeable	Changeable	Unchangeable	Unchangeable, but you can remove items and add new items
Indexed	Indexed	Indexed	Unindexed
Ordered	Unordered (In Python 3.6 and earlier) Ordered (from Python version 3.7)	Ordered	Unordered