Data Collections

Python has four collection data types: Lists, Tuples, Sets and Dictionaries. Of these Tuples and Sets are immutable data collections i.e. the items of these collections are not changeable.

TUPLES

A tuple is a collection which is ordered, unchangeable, and allows duplicate members. Unlike sets, lists and dictionaries, it is not possible to add elements to or remove elements from a tuple.

Tuples are created as follows using ()

```
tpl = (1,2,2,3)
```

Or

```
tpl = ("apple", "banana", "orange")
```

Or even

```
tpl = ("apple", "banana", 3, 4)
```

We could also have tuples inside a tuple (nested tuples) as follows:

```
tpl = ((1,2),(2,3))
```

Or

```
tpl = ((1,2),2)
```

Another way to create tuples is by using the `tuple()` constructor. This can be used to create tuples from lists, sets, dictionaries or strings. Here's an example:

```
s = {1,2,3} # This is a set
l = [1,2,3] # This is a list
d = {"brand": "Ford", "model": "Mustang"} # This is a dictionary
# sets, lists, and dictionaries will be covered in detail later
st = "hello"
print(tuple(s))
# Set is fundamentally unordered
# So the tuple can be created in any order of the set items
print(tuple(l))
print(tuple(d))
print(tuple(st))
```

The above code will output:

```
(2, 3, 1)
(1, 2, 3)
('brand', 'model')
('h', 'e', 'l', 'l', 'o')
```

The length of a tuple can be obtained using `len()`. An example is shown below:

```
tpl = (1,2,2,3)
print(len(tpl))
```

The above code will give the output: `4`

It is possible to use the membership operator `in` to check if an item is in a tuple. For example, `2 in (1,2,3)` will evaluate to `True`.

Accessing Tuple Items

Since tuple items are ordered, they can be accessed by their index. The first item has index 0 and the consecutive items have index 1, 2, 3 so on. It is also possible to use negative indices to access tuple items. The index -1 corresponds to the last element, -2 corresponds to the second last element and so on. Below is an example showing this.

```
tpl = ("apple", "banana", "mango", "orange")
print(tpl[0])
print(tpl[1])
print(tpl[-1])
print(tpl[-2])
```

The above code will output:

```
apple
banana
orange
mango
```

We can also take out slices of a tuple and get a new tuple by specifying where to start and end the slice. Below is an example showing this.

```
all_fruits = ("apple", "banana", "cherry", "kiwi", "mango", "orange")
some_fruits = all_fruits[2:5]
print(some_fruits)
```

In the above code, the range will start at index 2 (included) and end at index 5 (not included). The output of the code will be:

```
('cherry', 'kiwi', 'mango')
```

If we leave out the start value, the range will start at the first item and go till the given index (not included). For example,

```
("apple", "banana", "mango", "orange")[:2]
```

Will give the tuple `("apple", "banana")`

If we leave out the end value, the range will start at the given index (included) and go on till the end of the tuple. For example,

```
("apple", "banana", "mango", "orange")[2:]
```

Will give the tuple `("mango", "orange")`

We could also give negative indices as the start and end values. For example,

```
("apple", "banana", "mango", "orange")[-3:-1]
```

Will give the tuple `("banana", "mango")`

If the start index comes after the end index in a specified range, then the empty tuple is returned. For example, the following statements will give the empty tuple ( )

```
("apple", "banana", "mango", "orange")[3:2]
("apple", "banana", "mango", "orange")[-1:1]
```

It is also possible to loop through tuples using a `for` loop as follows:

```
for city in ("Bangalore", "Mysore", "Chennai")
   print(city)
```

The above code will output:

```
Bangalore
Mysore
Chennai
```

Unpacking Tuples

It is possible to unpack tuples as follows:

```
fruits = ("apple", "banana", "orange")
(fruit1, fruit2, fruit3) = fruits
# Equivalently, we could also unpack without the brackets () as below
# fruit1, fruit2, fruit3 = fruits
print(fruit1)
print(fruit2)
print(fruit3)
```

The above code will output:

```
apple
banana
orange
```

Tuple Methods

Python has two inbuilt functions that can be used on tuples. They are `count()` and `index()`.
`count()` returns the number of times a given argument occurs in a tuple. For example,

```
(1,3,2,5,3,4).count(3)
```

Will return `2`

`index()` returns the index of the first occurrence of a given argument in a tuple. For example,

```
(1,3,2,5,3,4).index(3)
```

Will return `1`

SETS

A set is a collection which is unordered, unchangeable (i.e. it is not possible to edit set elements) and disallows duplicate members. However, unlike tuples, it is possible to add elements to or remove elements from a set.

Sets are created as follows using {}

```
s = {1,2,3}
```

Or

```
s = {"apple", "banana", "orange"}
```

Or even

```
s = {"apple", "banana", 3, 4}
```

If duplicates are included while defining a set, they are considered only once.
Consider the below example:

```
s1 = {2,3,2,4,3,1,2,3}
s2 = {1,2,3,4}
print(s1==s2)
```

The above code will give the output: `True`

We could also have sets inside a set as follows:

```
s = {{1,2},{2,3}}
```

Or

```
s = {{1,2},2}
```

Note: The empty set is not {}, it is `set()` i.e. the `set()` constructor (explained next) acting on nothing. {} is an empty dictionary (which is dealt with in greater detail later).
The statement `type({})` will output: `<class 'dict'>`
But the statement `type(set())` will output: `<class 'set'>`

A set could also be created from tuples, lists, dictionaries and strings using the `set()` constructor. Here's an example:

```
t = (1,2,3) # tuple
l = [1,2,3] # list
d = {"brand": "Ford", "model": "Mustang"} # dictionary
s = "hello"
print(set(t))
print(set(l))
print(set(d))
print(set(s))
```

The above code will output:

```
{3, 2, 1}
{2, 3, 1}
{'model', 'brand'}
{'h', 'l', 'o', 'e'}
```

The length of a set can be obtained using `len()` function. An example is shown below:

```
s = {1,2,3,4}
print(len(s))
```

The above code will output: `4`

We can check if a specified item is in a set using the `in` operator. Below is an example:

```
s = {"apple", "orange", "banana"}
print("orange" in s)
```

The above code will output: `True`

Accessing Set Items

Since sets are unordered, their elements won't have any indices. Hence it is not possible to access set elements using an index like in tuples.

However it is possible to loop through the elements of a set using `for` loop.
An example is shown below:

```
for city in {"Bangalore", "Mysore", "Chennai"}
    print(city)
```

The above code will output:

```
Bangalore
Chennai
Mysore
```

Again due to the lack of a defined order for a set, every time the code is executed, the cities can be printed in a different order.

Adding and Removing Set Items

Items can be added to a set using the inbuilt method `add()`. Below is an example:

```
s = {"apple", "orange", "banana"}
s.add("mango")
print(s)
```

The above code will give the output: `{'apple', 'mango', 'banana', 'orange'}`

Items can be removed from a set using the inbuilt method `remove()`. Below is an example:

```
s = {"apple", "orange", "banana"}
s.remove("banana")
print(s)
```

The above code will give the output: `{'apple', 'orange'}`

Note: An alternate way to remove elements from a set is to use the `discard()` method. It works the same way as `remove()` with the difference being that `remove()` throws an error if the item to be removed does not exist in the set, however `discard()` won't throw an error in such a case.

More Set Methods

The union of two sets can be taken using the inbuilt function `union()`. Below is an example:

```
x = {"apple", "orange", "melon"}
y = {"melon", "mango", "kiwi"}
z = x.union(y)
print(z)
```

The above code will give the output: `{'apple', 'orange', 'melon', 'mango', 'kiwi'}`

The intersection of two sets can be taken using the inbuilt method `intersection()`
Below is an example:

```
x = {"apple", "orange", "banana", "mango"}
y = {"banana", "mango", "strawberry"}
z = x.intersection(y)
print(z)
```

The above code will give the output: `{'banana', 'mango'}`

The difference between two sets can be obtained using the inbuilt method `difference()`
Below is an example:

```
x = {"apple", "orange", "cherry", "banana"}
y = {"cherry", "banana", "melon"}
z = x.difference(y)
print(z)
```

The above code will give the output: `{'apple', 'orange'}`

LIST

A list is a collection which is ordered, changeable, and allows duplicate members. It is also possible to add elements to or remove elements from a list.

Lists are created as follows using `[]`

```
l = [1,2,2,3]
```

Or

```
l = ["apple", "banana", "orange"]
```

Or even

```
l = ["apple", "banana", 3, 4]
```

We could also have lists inside a list (nested lists) as follows:

```
l = [[1,2],[2,3]]
```

Or

```
l = [[1,2],2]
```

Another way to create lists is by using the `list()` constructor. This can be used to create lists from tuples, sets, dictionaries or strings. Here's an example:

```
t = (1,2,3) # tuple
s = {1,2,3} # set
d = {"brand": "Ford", "model": "Mustang"} # dictionary
st = "hello"
print(list(t))
print(list(s))
print(list(d))
print(list(st))
```

The above code will output:

```
[1, 2, 3]
[2, 3, 1]
['brand', 'model']
['h', 'e', 'l', 'l', 'o']
```

The length of a list can be obtained using `len()`. An example is shown below:

```
l = [1,2,2,3]
print(len(l))
```

The above code will give the output: `4`

To check if an item is in a list, we can use the membership operator `in`.
For example, `2 in [1,2,3]` will evaluate to `True`.

Accessing List Items

List items are ordered and they can be accessed by their index. The first item has index 0 and the consecutive items have index 1, 2, 3 so on. It is also possible to use negative indices to access list items. The index -1 corresponds to the last element, -2 corresponds to the second last element and so on. Below is an example showing this.

```
l = ["apple", "banana", "mango", "orange"]
print(l[2])
print(l[-1])
print(l[-3])
```

The above code will output:

```
mango
orange
banana
```

We can also take out slices of a list and get a new list by specifying where to start and end the slice. Below is an example showing this.

```
all_fruits = ["apple", "banana", "cherry", "kiwi", "mango", "orange"]
some_fruits = all_fruits[2:5]
print(some_fruits)
```

In the above code, the range will start at index `2` (included) and end at index `5` (not included).
The output of the code will be:

```
['cherry', 'kiwi', 'mango']
```

If we leave out the start value, the range will start at the first item and go till the given index (not included). For example,

```
["apple", "banana", "mango", "orange"][:2]
```

Will give the list `["apple", "banana"]`

If we leave out the end value, the range will start at the given index (included) and go on till the end of the list. For example,

```
["apple", "banana", "mango", "orange"][2:]
```

Will give the list `["mango", "orange"]`

We could also give negative indices as the start and end values. For example,

```
["apple", "banana", "mango", "orange"][-3:-1]
```

Will give the list `("banana", "mango")`

If the start index comes after the end index in a specified range, then the empty list is returned. For example, the following statements will give the empty list `[]`

```
["apple", "banana", "mango", "orange"][3:2]
["apple", "banana", "mango", "orange"][-1:1]
```

It is also possible to loop through lists using a `for` loop as follows:

```
for city in ["Bangalore", "Mysore", "Chennai"]
    print(city)
```

The above code will output:

```
Bangalore
Mysore
Chennai
```

Unpacking Lists

It is possible to unpack lists as follows:

```
fruits = ["apple", "banana", "orange"]
[fruit1, fruit2, fruit3] = fruits
# Equivalently, we could also unpack without the brackets [] as below
# fruit1, fruit2, fruit3 = fruits
print(fruit1)
print(fruit2)
print(fruit3)
```

The above code will output:

```
apple
banana
orange
```

Changing List Items

It is possible to change the list item at a particular index as follows:

```
l = [1,2,3]
l[1] = 5
print(l)
```

The above code will output: `[1, 5, 3]`

We could also change a range of list items as follows:

```
l = [1,2,3,4,5,6]
l[2:5] = [7,8,9]
print(l)
```

The above code will output: `[1, 2, 7, 8, 9, 6]`

While replacing a range of list with a new list as above, it is not necessary for the new list to have the same length as the length of the specified range. Consider the below example:

```
l1 = [1,2,3,4,5,6]
l2 = [1,2,3,4,5,6]
l1[2:5] = [7,8]
l2[1:3] = [7,8,9]
print(l1)
print(l2)
```

The above code will output:

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

We could insert an item to a list at a specified index, without altering the already existing list items by using the inbuilt method `insert()`. Below is an example:

```
l = ["apple", "banana", "mango"]
l.insert(2, "orange")
print(l)
```

The above code will output: `['apple', 'banana', 'orange', 'mango']`

To add an item at the end of a list we could use the inbuilt method `append()`
Below is an example:

```
l = ["apple", "banana", "cherry"]
l.append("watermelon")
print(l)
```

The above code will output: `['apple', 'banana', 'cherry', 'watermelon']`

We can also append a list at the end of a list using `extend()`. Below is an example:

```
l1 = ["apple", "banana", "mango"]
l2 = ["orange", "watermelon", "cherry"]
l1.extend(l2)
print(l1)
```

The above will give the output:

```
['apple', 'banana', 'mango', 'orange', 'watermelon', 'cherry']
```

Note: The argument of `extend()` need not necessarily be a list, it could be any data collection, so a tuple, set, dictionary or even a string would work. Below is an example with a tuple.

```
l1 = [1,2,3]
l2 = (4,5,6)
l1.extend(l2)
print(l1)
```

The above will give the output: `[1, 2, 3, 4, 5, 6]`

To remove a specific item from a list we could use the inbuilt method `remove()`
Below is an example:

```
l = ["apple", "banana", "cherry"]
l.remove("banana")
print(l)
```

The above code will output: `['apple', 'cherry']`

To remove an item from a specific index we could use the inbuilt method `pop()`
Below is an example:

```
l = ["apple", "banana", "cherry", "mango"]
l.pop(2)
print(l)
```

The above code will output: `['apple', 'banana', 'mango']`

When used without any arguments, `pop()` will remove the last item from the list.
Below is an example:

```
l = ["apple", "banana", "cherry", "mango"]
l.pop()
print(l)
```

The above code will output: `['apple', 'banana', 'cherry']`

Note: The item removed using `pop()` can be accessed as shown below.

```
l = ["apple", "banana", "cherry", "mango"]
a = l.pop(2)
print(l)
print(a)
```

The above code will give the output:

```
['apple', 'banana', 'mango']
'cherry'
```

To remove all items from a list, we could use the inbuilt method `clear()`
Below is an example:

```
l = [1,2,3]
l.clear()
print(l)
```

The above code will output: `[]`

More List Methods

`count()`
This is used to the count the number of times a given argument occurs in a list.
Example:

```
l = [1,3,2,4,3,5]
print(l.count(3))
```

Output: 2

`index()`
This is used to get the index of the first occurrence of a given argument in a list.
Example:

```
l = [1,3,2,4,3,5]
print(l.index(3))
```

Output: 1

`reverse()`
This can be used to reverse a list
Example:

```
l = [1,3,2,4,3,5]
l.reverse()
print(l)
```

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

`sort()`
This can be used to sort a list in ascending order.
Example:

```
l = [1,3,2,4,3,5]
l.sort()
print(l)
```

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

DICTIONARY

A dictionary is a collection which is ordered*, changeable and does not allow duplicates. It is possible to add items to or remove items from a dictionary. Dictionaries hold data in key:value pairs, and the values can be referred to by using the key name.

*as of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

Here's an example creation of a dictionary:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
```

Here' another example:

```
car = {
   "brand": "Ford",
   "electric": False,
   "year": 1964,
   "colours": ["red", "white", "blue"]
}
```

One more example:

```
d = {1:3, 4:2, 3:7}
```

As mentioned before, a dictionary does not allow duplicates. So there can't be more than one value for the same key (it is however possible to have the same value for different keys). If there are duplicates in the definition of a dictionary, then the key:value pair coming after will overwrite the key:value pair coming before. Below is an example of duplicates in a dictionary definition:

```
d = {1:3, 2:5, 2:4, 7:8}
print(d)
```

The above code will output: `{1: 3, 2: 4, 7: 8}`

The length of a dictionary can be obtained using `len()`. Below is an example:

```
d = {1:3, 2:5, 4:9, 7:8}
print(len(d))
```

The above code will output: `4`

The membership operator `in` will tell if a particular key is present in a dictionary.
For example, `"ab" in {"ab": "cd", "ef"; "gh"}` will give `True`.

Accessing Dictionary Items

As mentioned before, a dictionary holds data in key:value pairs. The value can be accessed by specifying the key. Below is an example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
print(car["model"])
```

The above code will output: `'Mustang'`

It is also possible to loop though the keys of a dictionary using a `for` loop as shown below:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
for key in car:
   print("Key: " + key + ", " + "Value: " + str(car[key]))
# String concatenation using + requires two strings
# But the value 1964 of the key 'year' is an integer
# Hence type casting is done using str()
```

The above code will give the output:

```
Key: brand, Value: Ford
Key: model, Value: Mustang
Key: year, Value: 1964
```

Changing Dictionary Items

The value of a key:value pair can be changed by referring to the key. Below is an example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car["year"] = 2021
print(car["year"])
```

The above code will output: 2021

If the key that is being referred to while assigning a value does not exist in the dictionary, then a new key:value pair will be created in the dictionary. Below is an example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car["electric"] = False
print(car)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'electric': False}
```

It is also possible to update an old dictionary with a new one using the `update()` method. If the new dictionary contains some keys that are also in the the old one, then the values corresponding to the intersecting keys of the old dictionary will be updated to the values in the new dictionary. Below is an example use of `update()`

```
car1 = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car2 = {
   "electric": False,
   "year": 2021,
   "colours": ["red", "white", "blue"]
}
car1.update(car2)
print(car1)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2021, 'electric': False,
'colours': ['red', 'white', 'blue']}
```

A specific key:value pair can be removed from the dictionary using `pop()`
Example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car.pop("year")
print(car)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang'}
```

Note: `pop()` also returns the value of the key:value pair that is popped. See the code below:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
a = car.pop("year")
print(car)
print(a)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang'}
1964
```

The last item* in a dictionary can be removed using `popitem()`
Example:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
car.popitem()
print(car)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang'}
```

*In Python 3.6 and earlier, dictionary is not ordered. Therefore in those versions, a random item is removed by `popitem()`

Note: `popitem()` also returns the key:value pair that is popped as a tuple. See the code below:

```
car = {
   "brand": "Ford",
   "model": "Mustang",
   "year": 1964
}
a = car.popitem()
print(car)
print(a)
```

The above code will output:

```
{'brand': 'Ford', 'model': 'Mustang'}
('year', 1964)
```

To remove all items from a dictionary, we could use the inbuilt method `clear()`
Below is an example:

```
d = {1:7, 2:5, 3:8}
d.clear()
print(d)
```

The above code will output: `{}`

STRINGS

Like many other popular programming languages, strings in Python are arrays of characters.
(An array is basically an ordered collection of elements of the same type.)
However, Python does not have a character data type. A single character is simply a string of length 1. But being array-like, many of the thing that we do with other iterables in Python like lists, tuples, sets can be done on strings too, as we will see in this section.

A string can be created by characters within single-quotation (') or double-quotation (") marks. For most of these notes, we have used double-quotation (") marks to represent strings. So from this section onwards we'll use single-quotation (') marks to represent strings.

The length of a string can obtained using `len()`. An example is shown below:

```
a = 'Python'
print(len(a))
```

The above code will output: `6`

The membership operator `in` can be used to check if a particular character or even a word is inside a string. For example,
`'a' in 'apple'` will return `True`,
`'app' in 'apple'` will return `True`,
`'b' in 'apple'` will return `False`

Just like lists and tuples, characters in a string can be accessed from an index. Example:

```
a = 'hello'
print(a[1])
```

The above code will output: `e`

We could also take out slices of a string like in lists and tuples. Example:

```
a = 'watermelon'
print(a[0:-5])
```

The above code will output: `water`

We can also loop though string characters using a `for` loop. Example:

```
for letter in ['bye']
   print(letter)
```

The above code will output:

```
b
y
e
```

String Methods

`upper()`
Converts all characters in string to upper case.
Example:

```
s = 'Python'
print(s.upper())
```

Output: `PYTHON`

`lower()`
Converts all characters in string to lower case.
Example:

```
s = 'Python'
print(s.lower())
```

Output: `python`

`strip()`
Removes all white spaces from before and after the text inside a string.
Example:
```
s = '   apple '
print(s == 'apple')
print(s.strip() == 'apple')
```
Output:
```
False
True
```

`replace()`
Replaces a substring with another string.
Example:
```
s = 'word'
s.replace('rd', 'rld')
print(s)
```
Output: `world`

Note: `replace()` won't modify the string if the given substring to be replaced is not present inside the string.

`split()`
Returns a list by splitting the string at the specified separator
Example:
```
s = 'a, b, c, d'
l = s.split(', ')
print(l)
```
Output: `['a', 'b', 'c', 'd']`

Note: If there is no argument given for `split()`, then the string will be split at whitespaces.
Example:
```
s = 'a b c d'
print(s.split())
```
Output: `['a', 'b', 'c', 'd']`

`count()`
Counts the number of times a specified character appear in a string.
Example:
```
s = 'apple'
print(s.count('p'))
```
Output: `2`

`index()`
Returns the index of the first appearance of a specified character.
Example:
```
s = 'apple'
print(s.index('p'))
```
Output: `1`

Formatting Strings

1. Using %

This is very similar to the `printf()` style formatting in C
Here's an example:

```
"My name is %s" %nm
```

In the above code, `%s` is replaced by the string stored in the variable `nm`
Consider the following code:

```
nm = "Suresh"
out = "My name is %s" %nm
print(out)
```

The above code would output: `My name is Suresh`

We could also do this with numeric literals.
We use `%d` for integers and `%f` for floating point numbers.
For floating point numbers, we can control the number of decimal places that are displayed by writing `%.<number of digits>f`. Here's an example:

```
"pi = %.3f" %3.14159
```

The above code will round off `3.14159` to 3 decimal places and display the output: `pi = 3.142`

Multiple substitutions can also be done as follows:

```
"There are %d students in %s college" %(num, name)
```

The above code will replace `%d` with the integer in `num`, and `%s` with the string in `name`

2. Using `format()`

This is a newer formatting method that is usually more preferred.
Here is an example of this style of formatting:

```
s = "My name is {}"
nm = "Suresh"
print(s.format(nm))
```

The above code will display the output: `My name is Suresh`

As can be seen above, in this formatting method, `{}` in a string is replaced by the argument of the `format()` function.

Here's another example:

```
pi = 3.1415
print("pi = {}".format(pi))
```

The above code will display the output: `pi = 3.1415`

So unlike the `%` method, here just `{}` works for both string and numeric literals.

Multiple substitutions can be done as follows:

```
"There are {} {} and {} {} in the basket".format(5,"apples",4,"oranges")
```

The above string will be formatted as:

```
"There are 5 apples and 4 oranges in the basket"
```

We can include indices in the parentheses to specify which argument is used for substitution.
Here's an example:

```
line = "There are {2} {0} and {2} {1} in the basket"
print(line.format("apples", "oranges", 5, 4))
```

The above code will output: `There are 5 apples and 5 oranges in the basket`