Introduction to Programming

Lab Worksheet

Week 7

Prior to attempting this lab tutorial ensure you have read the related lecture notes and/or viewed the lecture videos on MyBeckett. Once you have completed this lab you can attempt the associated exercises.

You can download this file in Word format if you want to make notes in it.

You can complete this work using the Python interpreter in interactive mode. This could be inside an IDE, or just a command prompt.

Topics covered:

- Introducing Sets
- Set Comprehensions
- Mutable and immutable Sets
- Set Operators and Methods
- Introduction Dictionaries
- Creating Dictionaries
- Working with Dictionaries

For more information about the module delivery, assessment and feedback please refer to the module within the MyBeckett portal.

Introducing Sets

The *Set* data-type is in many ways similar to a List and Tuple, except the contained values are not ordered and no duplicates are allowed. Also a set can only contain *immutable* values, which means that, for example, a set cannot contain list type values. A regular set is itself *mutable* however, so it can be changed after it has been created.

Since sets are not ordered, they do not support *indexing*, *slicing* or any method based on an element's position. However, they do have support for traditional mathematical style set operations such as *union*, *intersection* and *difference*.

Sets are very efficient at performing membership testing type operations, and much faster than Lists at this type of operation, especially when there are a large number of stored values. These sorts of differences explain why there are several similar data types available; the trick is often to pick whichever is best suited for some application.

The Set type is built directly into the Python language hence there is a specific syntax associated with creation and manipulation. So to create a Set directly, braces (curley brackets) can be used -

```
vowels = {"a", "e", "i", "o", "u"}
```

Remember ordering is not necessarily maintained and duplicate values are not allowed.

TASK: Try creating a set by entering the code below. Then use the `print()` function to display the contents of the set. Notice how the output varies from the entered values.

```
names = {"John", "Eric", "Terry", "Michael", "Graham", "Terry"}
>>> names = {"John", "Eric", "Terry", "Michael", "Graham", "Terry"}
>>>
>>> print(names)
{'Michael', 'John', 'Terry', 'Eric', 'Graham'}
>>>
>>>
```

A set can also be created by calling the `set()` *constructor*. A *constructor* is similar to a function, but is used to initialise an object based on the named type. However this constructor takes only a single parameter, which is *iterated* to extract the contents of the set. So to create the above set using the `set()` constructor the following code could be used:

```
names = set(["John", "Eric", "Terry", "Michael", "Graham", "Terry"])
```

Notice how a List was created (using `[ ]`) and then passed as a single parameter. If multiple parameters had been passed then an error would have been reported. Try entering the following and see the result -

```
names = set("John", "Eric", "Terry", "Michael", "Graham", "Terry")
```

Creating a set using the constructor is convenient if the values already exist in some other *iterable* value, such as a List or Tuple. It is even possible to create a set of individual characters by passing a string type value. This means that the following statements both create the exactly the same set:

```
hex_letters = {"a", "b", "c", "d", "e", "f"}

hex_letters = set("abcdef")
```

Finally an empty set must be created using the `set()` constructor rather than empty braces `{ }`, since the latter creates an empty *dictionary* (see later).

Set Comprehensions

The contents of a set can be created using a *set comprehension*, in the same way that a list comprehension can be used to create a list. In fact they work in exactly the same way, but appear between braces `{ }` rather than square brackets `[ ]`.

As with list comprehensions, the contents are created by evaluating an expression usually while *iterating* over a loop. For example, by using a set comprehension it is fairly easy to create a set which contains all the letters that appear within a specific `sentence` at least once. This uses the "uniqueness" property of Sets.

TASK: Enter the code below, then make a call to the `print()` function to display the contents of the set.

```
sentence = "this is a sentence containing some letters"
unique_letters = {x for x in sentence}
>>> sentence = "this is a sentence containing some letters"
>>> unique_letters = {x for x in sentence}
>>> print(unique_letters)
{'c', 'm', 'r', 't', 'i', 'o', 'n', 'e', 'l', ' ', 'g', 's', 'h',
'a'}
>>>
>>>
```

Remember that duplicates are not allowed, so each letter will appear in the set at most once. Also notice how a whitespace character has been included into the produced set.

Set comprehensions can also restrict the included values, by appending an `if` statement. So to create the same as the above, while excluding whitespaces the following variation could be used:

```
unique_letters = {x for x in sentence if x != " "}
```

Set Operations

One of the most common operations performed on a set is membership testing. This is done using either the `in` or `not in` operators. These check whether a specific element is either present, or not present, within a given set. For example:

```
name = input("Enter your name: ")
if name in names:
     print("You are listed in the set of known names")
```

Even sets containing many thousands of elements can be checked almost instantly.

TASK: Rewrite the previous code so that it checks that the input name is NOT within the set of known names. Hint: use the `not in` operator.

```
>>> known_names = {"John", "Eric", "Terry", "Michael", "Graham"}
>>>
>>> input_name = input("Enter a name: ")
Enter a name:
>>> if input_name not in known_names:
...     print(f"{input_name} is not in the set of known names.")
... else:
...     print(f"{input_name} is in the set of known names.")
...
 is not in the set of known names.
>>>
```

Since a set type represents an existing well known concept, a mathematical set, there are certain operations that can be performed using both *methods* and *operators*. Since sets are mutable, both ***accessor*** and ***mutator*** type operations exist. It is also possible to perform special ***comparison*** type operations on sets.

The typical mathematical style operations that can be performed include *union*, *intersection*, *difference*, and *symmetric difference*.

The operators used to support these are as follows -

`|`	*union (the symbol is a "pipe", usually found to the left of the Z key)*
`&`	*intersection*
`–`	*difference*
`^`	*symmetric difference*

Each of these can also be achieved by calling an equivalent *method*. Fortunately the method names closely reflect the mathematical operation, and the methods are called `union()`, `intersection()`, `difference()` and `symmetric_difference()`.

TASK: Use the built-in `help()` function to view all the methods available on the `set` type.

```
>>> help(set)

Help on class set in module builtins:


class set(object)

 |  set() -> new empty set object

 |  set(iterable) -> new set object

 |

 |  Build an unordered collection of unique elements.

 |

 |  Methods defined here:

 |

 |  __and__(self, value, /)

 |      Return self&value.

 |

 |  __contains__(...)

 |      x.__contains__(y) <==> y in x.

 |

 |  __eq__(self, value, /)

 |      Return self==value.

 |

 |  __ge__(self, value, /)

 |      Return self>=value.

 |

 |  __getattribute__(self, name, /)

 |      Return getattr(self, name).
```

```
|
| __gt__(self, value, /)
|     Return self>value.
|
| __iand__(self, value, /)
|     Return self&=value.
-- More  --
```

If we assume we have the following sets:

```
staff = {"Pete", "Kelly", "Jon", "Paul", "Sally", "Sue"}
directors = {"Kelly", "Rupert", "Cyril", "Jon"}
```

We could use the *union* operator `|` to add to the members of a se:

```
>>> staff = staff | {"Mark", "Ringo"}
>>> print(staff)

{'Kelly', 'Paul', 'Jon', 'Sally', 'Ringo', 'Pete', 'Mark', 'Sue'}
```

We could use the *intersection* operator `&` to find only common elements:

```
>>> staff_directors = staff & directors
>>> print(staff_directors)

{'Kelly', 'Jon'}
```

We could use the *difference* operator `-` to find elements in one set, but not the other:

```
>>> external = directors - staff
>>> print(external)

{'Cyril', 'Rupert'}
```

We could use the *symmetric difference* operator `^` to find elements in either set, but not in both:

```
>>> staff_or_external = directors ^ staff
>>> print(staff_or_external)

{'Cyril', 'Paul', 'Sally', 'Pete', 'Rupert', 'Sue', 'Ringo', 'Mark'}
```

TASK: Create the two initial sets, `staff` and `directors` as shown in the first example above. Perform the four mathematical set operations shown, but use the equivalent *method* calls to achieve the same results. For example:

```
staff = staff | {"Mark", "Ringo"}

# becomes …

staff = staff.union({"Mark", "Ringo"})

>>> staff = {"Pete", "Kelly", "Jon", "Paul", "Sally", "Sue"}
>>> directors = {"Kelly", "Rupert", "Cyril", "Jon"}
>>>
>>> staff_union_directors = staff.union(directors)
>>> staff_intersection_directors = staff.intersection(directors)
>>> staff_difference_directors = staff.difference(directors)
>>> staff_symmetric_difference_directors = staff.symmetric_difference(directors)
>>>
>>> print("Union:", staff_union_directors)
Union: {'Sue', 'Jon', 'Pete', 'Paul', 'Sally', 'Kelly', 'Rupert', 'Cyril'}
>>> print("Intersection:", staff_intersection_directors)
Intersection: {'Kelly', 'Jon'}
>>> print("Difference:", staff_difference_directors)
Difference: {'Sue', 'Sally', 'Pete', 'Paul'}
>>> print("Symmetric Difference:", staff_symmetric_difference_directors)
Symmetric Difference: {'Sue', 'Pete', 'Paul', 'Sally', 'Rupert', 'Cyril'}
>>>
>>>
```

The set operations we have seen so far have all been **accessor** type methods; they do not directly change the set to which the operation was applied, hence the reason the above examples all assigned the result to a variable. However, the same set of mathematical operations can be applied as **mutators**, so they do directly change the contents of the set.

If the same operators we have seen are applied using the *Augmented Assignment* style, then they act as mutators. The following statement removes the given elements from the set directly, using an augmented difference operator.

```
staff -= {"Mark", "Ringo"}
```

The same mutator type behaviour can also be achieved using the methods, however different versions of the methods need to be called. The *mutator* versions of the methods are called `update()`, `intersection_update()`, `difference_update()` and `symmetric_difference_update()`.

TASK: Create the set shown below then use the *mutator* version of the union method, which is `update()` to add the two missing vowels to the set.

```
vowels = set({"a", "e", "i"})

>>> vowels = {"a", "e", "i"}
>>>
>>> vowels.update({"o", "u"})
>>>
>>> print(vowels)
{'u', 'i', 'a', 'o', 'e'}
>>>
```

Set Comparison Operations

As well as the *accessor* and *mutator* type operations, there are also set specific **comparisons** available. Like the previous set of operations, these are based on well known mathematical style operations.

The comparison operations that can be performed are *subset*, *proper subset*, *superset,* and *proper superset*. These return a `True` or `False` result.

The operators used to support these are as follows -

<=	*subset*
<	*proper subset*
>=	*superset*
>	*proper superset*

If we assume we have the following sets -

```
staff = {"Pete", "Kelly", "Jon", "Paul", "Sally", "Sue"}
managers = {"Kelly", "Jon", "Paul", "Sally", "Sue"}
```

We could use the *subset* operator < to check whether all the elements of the `managers` set are also contained within the `staff` set.

```
if managers < staff:
      print("All managers are staff members")
```

The *proper subset* operator <= does the same check, but also ensures that the sets are not exactly equal, in other words a set is *strictly* a subset of another set. The *superset* > and *proper superset* >= operators do the reverse checks. So the following is equivalent to the previous example:

```
if staff > managers:
      print("All managers are staff members")
```

There are also methods available for performing the same type of comparisons, which are called `issubset()` and `issuperset()`.

TASK: Write code based on the previous two examples, but use the equivalent *method* calls to achieve the same results.

```
>>> staff = {"Pete", "Kelly", "Jon", "Paul", "Sally", "Sue"}

>>> directors = {"Kelly", "Rupert", "Cyril", "Jon"}

>>>

>>> staff_union_directors = staff.union(directors)

>>> staff_intersection_directors = staff.intersection(directors)

>>> staff_difference_directors = staff.difference(directors)

>>> staff_symmetric_difference_directors = staff.symmetric_difference(directors)

>>>

>>> print("Union:", staff_union_directors)

Union: {'Pete', 'Jon', 'Sally', 'Kelly', 'Paul', 'Cyril', 'Rupert', 'Sue'}

>>> print("Intersection:", staff_intersection_directors)

Intersection: {'Kelly', 'Jon'}

>>> print("Difference:", staff_difference_directors)

Difference: {'Pete', 'Sue', 'Paul', 'Sally'}

>>> print("Symmetric Difference:", staff_symmetric_difference_directors)

Symmetric Difference: {'Pete', 'Sally', 'Paul', 'Cyril', 'Rupert', 'Sue'}

>>>
```

Creating an Immutable Set

As we have already seen, when a set is constructed using the braces `{ }`, or via the `set()` constructor it is *mutable*. It is possible however to create an *immutable* set also. This is done by using the `frozenset()` constructor. It looks the same as the other constructor:

```
suits = frozenset({"heart", "club", "spade", "diamond"})
```

If a set is created in this way then its contents cannot be changed. The operators and methods available generally remain the same as with a regular set, however anything that *mutates* the contents is not available. So for example, the `update()` method is not available on a frozenset. However all of the *accessor* and *comparison* operators work in exactly the same way as a regular set.

```
>>> {"club", "diamond"} < suits
True
```

TASK: Use the built-in `help()` function to view all the methods available on the `frozenset` type.

```
>>> help(frozenset)

Help on class frozenset in module builtins:


class frozenset(object)

 |  frozenset() -> empty frozenset object

 |  frozenset(iterable) -> frozenset object

 |

 |  Build an immutable unordered collection of unique elements.

 |

 |  Methods defined here:

 |

 |  __and__(self, value, /)

 |      Return self&value.

 |

 |  __contains__(...)

 |      x.__contains__(y) <==> y in x.

 |

 |  __eq__(self, value, /)

 |      Return self==value.

 |

 |  __ge__(self, value, /)

 |      Return self>=value.

 |
 |  __getattribute__(self, name, /)

 |      Return getattr(self, name).

 |

 |  __gt__(self, value, /)

 |      Return self>value.

 |

 |  __hash__(self, /)

-- More  --
```

Introducing Dictionaries

The *Dictionary* data-type stores multiple values like the other collection types. However, what is distinct about a dictionary is that it stores elements as pairs, often called a *key:value* pair. Dictionaries are *ordered* and *mutable* and can have *key:value* pairs added and removed after initial creation.

Each *key* is unique and is associated with a single value. i.e. a *key* **maps** to a *value* just like a *word* in a conventional language dictionary **maps** to a *definition*. The set of keys must be unique - each key can appear at most once. However the values in the dictionary do not need to be unique, therefore different keys can be mapped to the same value.

Dictionaries are useful when a value needs to be located quickly given a known key. For example a dictionary may store a collection of *customer* records (values), and the key to these is the unique *customer number*.

Since the Dictionary type is built directly into the Python language there is a specific syntax associated with creation and manipulation. This is similar to what we have already seen with a Set. The main difference however is that operations involving Dictionaries usually involve pairs of values. To create a Dictionary directly, braces (curly brackets) can be used:

```
stock = {"apple":10, "banana":15, "orange":11}
```

This is similar to the notation for a Set, but the contents of the brackets show that it is a dictionary being created. Notice that a pair is specified for each entry separated by a colon ‘`:`’, in the form *key:value*.

The *key* of the first element within the example is the string `"apple"`, and the *value* associated with this is the number `10`. Unlike a set the insertion ordering of the elements is maintained (from Python version 3.7 at least).

A dictionary can also be created by calling the `dict()` *constructor*. This can be called with various types of arguments and is somewhat more flexible than the `set()` constructor. To create the above dictionary, any of the following variations could be used:

```
# create by passing a dictionary
stock = dict({"apple":10, "banana":15, "orange":11})

# create by passing keywords (only possible if keys are strings)
stock = dict(apple=10, banana=15, orange=11)

# create by passing list of tuples
stock = dict([("apple",10), ("banana",15), ("orange",11)])
```

Finally an empty dictionary can be created using either empty braces `{ }` or the `dict()` constructor with no arguments. This explains why, as mentioned earlier, an empty *Set* can ONLY be created using the `set()` constructor with no arguments.

Dictionary Comprehensions

As with sets and lists, it is also possible to create a dictionary using a *comprehension*. A very slightly different expression syntax is required, since a pair of values need to be provided for each generated entry. This is handled by the inclusion of a '`:`' to separate the *key* from the *value*.

TASK: Enter the code below, then make a call to the `print()` function to display the contents of the dictionary.

```
import math

roots = {n : math.sqrt(n) for n in range(1,26)}

>>> import math
>>>
>>> roots = {n: math.sqrt(n) for n in range(1, 26)}
>>>
>>> print(roots)
{1: 1.0, 2: 1.4142135623730951, 3: 1.7320508075688772, 4: 2.0, 5:
2.23606797749979, 6: 2.449489742783178, 7: 2.6457513110645907, 8:
2.8284271247461903, 9: 3.0, 10: 3.1622776601683795, 11:
3.3166247903554, 12: 3.4641016151377544, 13: 3.605551275463989, 14:
3.7416573867739413, 15: 3.872983346207417, 16: 4.0, 17:
4.123105625617661, 18: 4.242640687119285, 19: 4.358898943540674, 20:
4.47213595499958, 21: 4.58257569495584, 22: 4.69041575982343, 23:
4.795831523312719, 24: 4.898979485566356, 25: 5.0}
>>>
```

This produces a dictionary that maps a numeric value (in the range 1..25), to its square root.

Hence, the created dictionary will have a total of 25 entries. In which each *key* is an integer type that maps to a float type.

Manipulating Dictionaries

Once a dictionary has been created there are several mechanisms available to access and mutate its content. Remember, most operations rely on the *key* being specified. For example, it is possible to access a *value* from the dictionary by providing its *key* as an *index*:

```
print("Apple stock level is", stock["apple"])
```

Notice how the provided `[ index ]` value is not necessarily an integer, as with lists, strings or tuples. The provided index value MUST match the data-type of the dictionaries key set, in this case a string.

Since dictionaries are mutable, it is also possible to add or update existing *key:value* pairs using the indexing notation:.

```
stock["pear"] = 50            # add new key:value pair
stock["apple"] += 1           # increase apple stock level
```

TASK: Write some code which adds a new fruit called `"kiwi"` to the `stock` dictionary, with an initial stock level of `10`.

>>> stock = {"apple": 20, "banana": 15, "orange": 30}

>>>

>>> stock["kiwi"] = 10

>>>

>>> print(stock)

{'apple': 20, 'banana': 15, 'orange': 30, 'kiwi': 10}

>>>

As with sets a common operation performed on dictionaries is membership testing. This can be done using exactly the same syntax as sets, but only the *key* needs providing, e.g.

```
if "apple" in stock:
     print("Apples have a stock level")
```

Dictionary Methods

Many of the *methods* which are available on the other collection data-types, such as lists, are also available on dictionaries. These include `clear()`, `copy()`, `get()`, `pop()` and `update()`.

These all typically perform in the same away as with the other data-types, but in some cases take slightly different parameters involving a *key* and *value*.

```
# pop the "orange" returning its stock level
stock.pop("orange")

# update the stock to include two new fruits
stock.update(lemmon=15, strawberry=99)
```

TASK: Use the built-in `help()` function to view all the methods available on the `dict` type. Then write some code that uses the `popitem()` method to remove some *key:value* pairs from the `stock` dictionary.

```
>>> stock = {"apple": 20, "banana": 15, "orange": 30}

>>>

>>> help(dict)

Help on class dict in module builtins:



class dict(object)

 |  dict() -> new empty dictionary

 |  dict(mapping) -> new dictionary initialized from a mapping object's

 |      (key, value) pairs

 |  dict(iterable) -> new dictionary initialized as if via:

 |      d = {}

 |      for k, v in iterable:

 |          d[k] = v

 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs

 |      in the keyword argument list.  For example:  dict(one=1, two=2)

 |
 |  Methods defined here:

 |

 |  __contains__(self, key, /)

 |      True if the dictionary has the specified key, else False.

 |

 |  __delitem__(self, key, /)

 |      Delete self[key].

 |

 |  __eq__(self, value, /)

 |      Return self==value.

 |

 |  __ge__(self, value, /)

 |      Return self>=value.

 |

 |  __getattribute__(self, name, /)

 |      Return getattr(self, name).

 |

 |  __getitem__(self, key, /)

 |      Return self[key].

 |

 |  __gt__(self, value, /)

 |      Return self>value.

 |

 |  __init__(self, /, *args, **kwargs)

 |      Initialize self.  See help(type(self)) for accurate signature.

 |
|  __ior__(self, value, /)

|      Return self|=value.

|

|  __iter__(self, /)

|      Implement iter(self).

|

|  __le__(self, value, /)

|      Return self<=value.

|

|  __len__(self, /)

|      Return len(self).

|

|  __lt__(self, value, /)

|      Return self<value.

|

|  __ne__(self, value, /)

|      Return self!=value.

|

|  __or__(self, value, /)

|      Return self|value.

|

|  __repr__(self, /)

|      Return repr(self).

|

|  __reversed__(self, /)

|      Return a reverse iterator over the dict keys.
```

```
 |
 |  __ror__(self, value, /)
 |      Return value|self.
 |
 |  __setitem__(self, key, value, /)
 |      Set self[key] to value.
 |
 |  __sizeof__(...)
 |      D.__sizeof__() -> size of D in memory, in bytes
 |
 |  clear(...)
 |      D.clear() -> None.  Remove all items from D.
 |
 |  copy(...)
 |      D.copy() -> a shallow copy of D
 |
 |  get(self, key, default=None, /)
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  items(...)
 |      D.items() -> a set-like object providing a view on D's items
 |
 |  keys(...)
 |      D.keys() -> a set-like object providing a view on D's keys
 |
 |  pop(...)
 |      D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
 |
 |      If the key is not found, return the default if given; otherwise,
 |      raise a KeyError.
 |
 |  popitem(self, /)
 |      Remove and return a (key, value) pair as a 2-tuple.
 |
 |      Pairs are returned in LIFO (last-in, first-out) order.
 |      Raises KeyError if the dict is empty.
 |
 |  setdefault(self, key, default=None, /)
 |      Insert key with a value of default if key is not in the dictionary.
 |
 |      Return the value for key if key is in the dictionary, else default.
 |
 |  update(...)
 |      D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
 |      If E is present and has a .keys() method, then does:  for k in E: D[k] = E[k]
 |      If E is present and lacks a .keys() method, then does:  for k, v in E: D[k] = v
 |      In either case, this is followed by: for k in F:  D[k] = F[k]
 |
 |  values(...)
 |      D.values() -> an object providing a view on D's values
 |
 |  ----------------------------------------------------------------------
```

| Class methods defined here:

-- More --

Iterating over Dictionaries

It is very common to *iterate* over the contents of a dictionary, using a `for...in` type loop.

If just the name of the dictionary is provided, then only the *key* is extracted during each iteration. So the following would print the *keys* of the dictionary only:

```
for item in stock:
     print(item)

apple
banana
orange
```

It is possible to iterate over the *values* only by calling the `values()` method, e.g.

```
for level in stock.values():
     print(level)

10
15
11
```

Finally it is possible to iterate over each *key:value* pair. To do this the `items()` method can be used. This allows access to each key and value as a *Tuple*, hence each pair can be extracted using *sequence unpacking.* The code is:

```
for item,level in stock.items():
     print(item, "has a stock level of", level)

apple has a stock level of 10
banana has a stock level of 15
orange has a stock level of 11
```

TASK: Write some code that iterates over the contents of the `roots` dictionary created within an earlier task. For each entry, print the message -

“`The square root of <num> is <sqrt>`”

Where `<num>` shows the number, and `<sqrt>` shows the square root of that number.

```
>>> import math

>>>
```

```
>>> roots = {n: math.sqrt(n) for n in range(1, 26)}

>>>

>>> for num, sqrt in roots.items():

...     print(f"The square root of {num} is {sqrt}")

...

The square root of 1 is 1.0

The square root of 2 is 1.4142135623730951

The square root of 3 is 1.7320508075688772

The square root of 4 is 2.0

The square root of 5 is 2.23606797749979

The square root of 6 is 2.449489742783178

The square root of 7 is 2.6457513110645907

The square root of 8 is 2.8284271247461903

The square root of 9 is 3.0

The square root of 10 is 3.1622776601683795

The square root of 11 is 3.3166247903554

The square root of 12 is 3.4641016151377544

The square root of 13 is 3.605551275463989

The square root of 14 is 3.7416573867739413

The square root of 15 is 3.872983346207417

The square root of 16 is 4.0

The square root of 17 is 4.123105625617661

The square root of 18 is 4.242640687119285

The square root of 19 is 4.358898943540674

The square root of 20 is 4.47213595499958

The square root of 21 is 4.58257569495584
The square root of 22 is 4.69041575982343

The square root of 23 is 4.795831523312719

The square root of 24 is 4.898979485566356

The square root of 25 is 5.0

>>>
```

---

ey Terminology

Understanding key terminology is important not only when programming but for understanding computer science in general. Certain terms, technologies and phrases will often re-occur when reading or learning about many computer science topics. Having an awareness of these is important, as it makes learning about the various subjects and communication with others in the field easier.

TASK: Look at each of the phrases below and ensure you understand what each of these means. For any that you do not understand, do a little research to find a definition of each term. This research may involve looking back over these notes, or the associated lecture notes. It may also involve searching for these terms on the Internet.

- Set
- Set operations
- Set comprehension
- Dictionary
- *key:value* pair

Set: A set is an unordered collection of distinct elements. Elements are enclosed in curly brackets {} to define sets. The elements' order is not guaranteed, and duplicate values are not permitted.

Set operations: Different mathematical operations that can be carried out on sets are referred to as set operations. Union, intersection, difference, and symmetric difference are examples of common set operations.

Set comprehension: A single line of code can be used to build sets in using set comprehension. It entails utilizing a single-line expression enclosed in curly braces {} to specify a set and its elements.

Dictionary: An unordered collection of key-value pairs is called a dictionary. Every key-value pair associates a key with a corresponding value. Curly braces {} are used to define dictionaries, and colons : are used to divide keys and values.

Key:value pair: A key-value pair is a basic data structure found in dictionaries, where a distinct key is linked to a particular value to facilitate effective information storing and retrieval.

Practical Exercises

You have now completed this tutorial. Now attempt to complete the associated exercises.