1

Python Data Types and Structures

Data structures and algorithms are important components in the development of any software system. An algorithm can be defined as a set of step-by-step instructions to solve any given problem; an algorithm processes the data and produces the output results based on the specific problem. The data used by the algorithm to solve the problem has to be stored and organized efficiently in the computer memory for the efficient implementation of the software. The performance of the system depends upon the efficient access and retrieval of the data, and that depends upon how well the data structures that store and organize the data in the system are chosen.

Data structures deal with how the data is stored and organized in the memory of the computer that is going to be used in a program. Computer scientists should understand how efficient an algorithm is and which data structure should be used in its implementation. The Python programming language is a robust, powerful, and widely used language to develop software-based systems. Python is a high-level, interpreted, and object-oriented language that is very convenient to learn and understand the concepts of data structures and algorithms.

In this chapter, we briefly review the Python programming language components that we will be using to implement the various data structures discussed in this book. For a more detailed discussion on the Python language in broader terms, take a look at the Python documentation:

- `https://docs.python.org/3/reference/index.html`
- `https://docs.python.org/3/tutorial/index.html`

In this chapter, we will look at the following topics:

- Introducing Python 3.10
- Installing Python
- Setting up a Python development environment
- Overview of data types and objects
- Basic data types
- Complex data types
- Python's collections module

Introducing Python 3.10

Python is an interpreted language: the statements are executed line by line. It follows the concepts of object-oriented programming. Python is dynamically typed, which makes it an ideal candidate among languages for scripting and fast-paced development on many platforms. Its source code is open source, and there is a very big community that is using and developing it continuously, at a very fast pace. Python code can be written in any text editor and saved with the `.py` file extension. Python is easy to use and learn because of its compactness and elegant syntax.

Since the Python language will be used to write the algorithms, an explanation is provided of how to set up the environment to run the examples.

Installing Python

Python is preinstalled on Linux- and Mac-based operating systems. However, you will want to install the latest version of Python, which can be done on different operating systems as per the following instructions.

Windows operating system

For Windows, Python can be installed through an executable `.exe` file.

1. Go to `https://www.python.org/downloads/`.
2. Choose the latest version of Python—currently, it is 3.10.0—according to your architecture. If you have a 32-bit version of Windows, choose the 32-bit installer; otherwise, choose the 64-bit installer.
3. Download the `.exe` file.
4. Open the `python-3.10.0.exe` file.

5. Make sure to check **Add Python 3.10.0 to PATH**.
6. Click **Install Now** and then wait until the installation is complete; you can now use Python.
7. To verify that Python is installed correctly, open the Command Prompt and type the `python --version` command. It should output `Python 3.10.0`.

Linux-based operating systems

To install Python on a Linux machine, take the following steps:

1. Check whether you have Python preinstalled by entering the `python --version` command in the terminal.
2. If you do have not a version of Python, then install it through the following command:

   ```
   sudo apt-get install python3.10
   ```

3. Now, verify that you have installed Python correctly by typing the `python3.10 --version` command in the terminal. It should output `Python 3.10.0`.

Mac operating system

To install Python on a Mac, take the following steps:

1. Go to `https://www.python.org/downloads/`.
2. Download and open the installer file for `Python 3.10.0`.
3. Click **Install Now**.
4. To verify that Python is installed correctly, open the terminal and type `python –version`. It should output `Python 3.10.0`.

Setting up a Python development environment

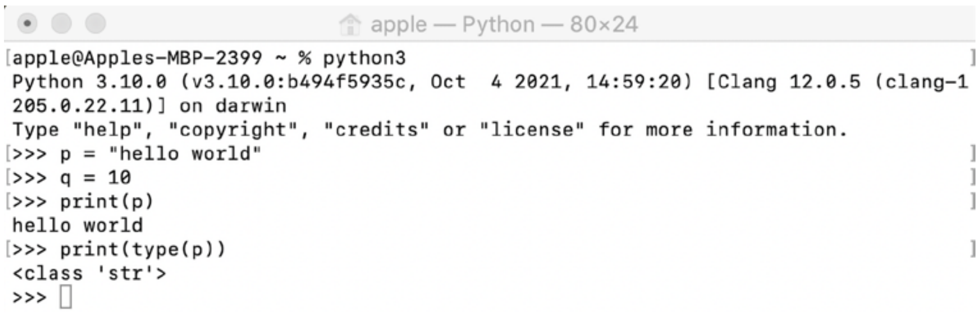
Once you have installed Python successfully for your respective OS, you can start this hands-on approach with data structures and algorithms. There are two popular methods to set up the development environment.

Setup via the command line

The first method to set up the Python executing environment is via the command line, after installation of the Python package on your respective operating system. It can be set up using the following steps.

1. Open the terminal on Mac/Linux OS or Command Prompt on Windows.

2. Execute the Python 3 command to start Python, or simply type py to start Python in the Windows Command Prompt.
3. Commands can be executed on the terminal.



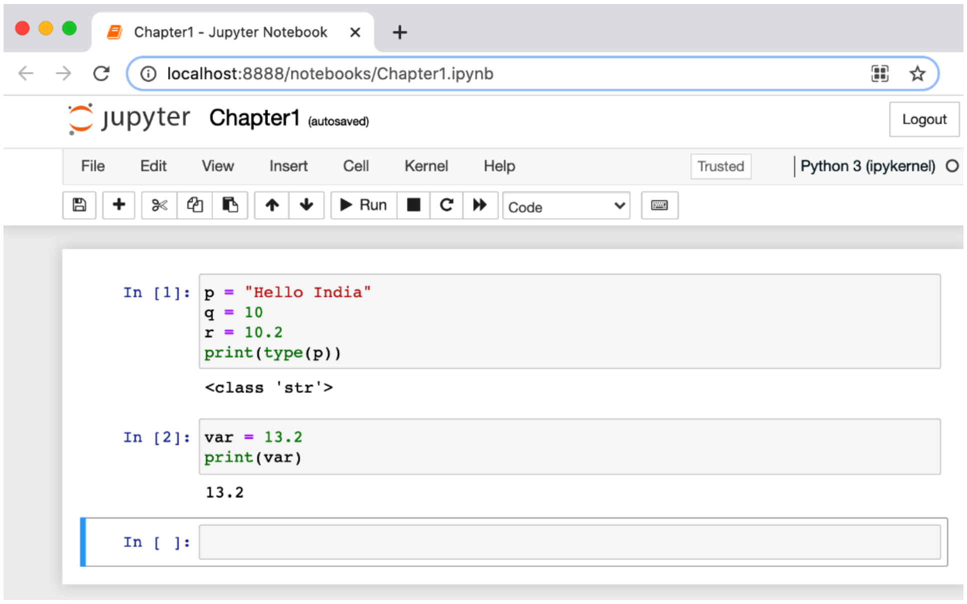
```
apple@Apples-MBP-2399 ~ % python3
Python 3.10.0 (v3.10.0:b494f5935c, Oct  4 2021, 14:59:20) [Clang 12.0.5 (clang-1
205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p = "hello world"
>>> q = 10
>>> print(p)
hello world
>>> print(type(p))
<class 'str'>
>>>
```

Figure 1.1: Screenshot of the command-line interface for Python

The User Interface for the command-line execution environment is shown in *Figure 1.1*.

Setup via Jupyter Notebook

The second method to run the Python program is through Jupyter Notebook, which is a browser-based interface where we can write the code. The User Interface of Jupyter Notebook is shown in *Figure 1.2*. The place where we can write the code is called a "cell."


Chapter1 - Jupyter Notebook
localhost:8888/notebooks/Chapter1.ipynb
jupyter Chapter1 (autosaved)
Logout
File Edit View Insert Cell Kernel Help
Trusted
Python 3 (ipykernel)
Run
Code

```
In [1]: p = "Hello India"
        q = 10
        r = 10.2
        print(type(p))

        <class 'str'>

In [2]: var = 13.2
        print(var)

        13.2

In [ ]:
```

Figure 1.2: Screenshot of the Jupyter Notebook interface

Once Python is installed, on Windows, Jupyter Notebook can be easily installed and set up using a scientific Python distribution called Anaconda by taking the following steps.

1. Download the Anaconda distribution from `https://www.anaconda.com/products/individual`.
2. Install it according to the installation instructions.
3. Once installed, on Windows, we can run the notebook by executing the `jupyter notebook` command at the Command Prompt. Alternatively, following installation, the `Jupyter Notebook` app can be searched for and run from the taskbar.
4. On Linux/Mac operating systems, Jupyter Notebook can be installed using `pip3` by running the following code in the terminal:

   ```
   pip3 install notebook
   ```

5. After installation of Jupyter Notebook, we can run it by executing the following command at the Terminal:

   ```
   jupyter notebook
   ```

   On some systems, this command does not work, depending upon the operating system or system configuration. In that case, Jupyter Notebook should start by executing the following command on the terminal.

   ```
   python3 -m notebook
   ```

It is important to note that we will be using Jupyter Notebook to execute all the commands and programs throughout the book, but the code will also function in the command line if you'd prefer to use that.

Overview of data types and objects

Given a problem, we can plan to solve it by writing a computer program or software. The first step is to develop an algorithm, essentially a step-by-step set of instructions to be followed by a computer system, to solve the problem. An algorithm can be converted into computer software using any programming language. It is always desired that the computer software or program be as efficient and fast as possible; the performance or efficiency of the computer program also depends highly on how the data is stored in the memory of a computer, which is then going to be used in the algorithm.

The data to be used in an algorithm has to be stored in variables, which differ depending upon what kind of values are going to be stored in those variables. These are called *data types*: an integer variable can store only integer numbers, and a float variable can store real numbers, characters, and so on. The variables are containers that can store the values, and the values are the contents of different data types.

In most programming languages, variables and their data types must initially be declared, and then only that type of data can be statically stored in those variables. However, in Python, this is not the case. Python is a dynamically typed language; the data type of the variables is not required to be explicitly defined. The Python interpreter implicitly binds the value of the variable with its type at runtime. In Python, data types of the variable type can be checked using the function `type()`, which returns the type of variable passed. For example, if we enter the following code:

```python
p = "Hello India"
q = 10
r = 10.2
print(type(p))
print(type(q))
print(type(r))
print(type(12+31j))
```

We will get an output like the following:

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'complex'>
```

The following example, demonstrates a variable that has a `var` float value, which is substituted for a string value:

```python
var = 13.2
print(var)

print(type (var))

var = "Now the type is string"
print(type(var))
```

The output of the code is:

```
13.2
<class 'float'>
<class 'str'>
```

In Python, every item of data is an object of a specific type. Consider the preceding example; here, when a variable `var` is assigned a value of `13.2`, the interpreter initially creates a float object having a value of `13.2`; a variable `var` then points to that object as shown in *Figure 1.3*:

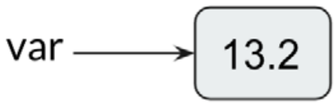


Figure 1.3: Variable assignment

Python is an easy-to-learn object-oriented language, with a rich set of built-in data types. The principal built-in types are as follows and will be discussed in more detail in the following sections:

- Numeric types: `Integer (int)`, `float`, `complex`
- Boolean types: `bool`
- Sequence types: `String (str)`, `range`, `list`, `tuple`
- Mapping types: `dictionary (dict)`
- Set types: `set`, `frozenset`

We will divide these into basic (numeric, Boolean, and sequence) and complex (mapping and set) data types. In subsequent sections, we will discuss them one by one in detail.

Basic data types

The most basic data types are numeric and Boolean types. We'll cover those first, followed by sequence data types.

Numeric

Numeric data type variables store numeric values. Integer, float, and complex values belong to this data type. Python supports three types of numeric types:

- **Integer (int)**: In Python, the interpreter takes a sequence of decimal digits as a decimal value, such as the integers `45`, `1000`, or `-25`.

- **Float:** Python considers a value having a floating-point value as a float type; it is specified with a decimal point. It is used to store floating-point numbers such as `2.5` and `100.98`. It is accurate up to 15 decimal points.
- **Complex**: A complex number is represented using two floating-point values. It contains an ordered pair, such as a + *i*b. Here, a and b denote real numbers and *i* denotes the imaginary component. The complex numbers take the form of `3.0 + 1.3i`, `4.0i`, and so on.

Boolean

This provides a value of either `True` or `False`, checking whether any statement is true or false. `True` can be represented by any non-zero value, whereas `False` can be represented by `0`. For example:

```
print(type(bool(22)))
print(type(True))
print(type(False))
```

The output will be the following:

```
<class 'bool'>
<class 'bool'>
<class 'bool'>
```

In Python, the numeric values can be used as bool values using the built-in `bool()` function. Any number (integer, float, complex) having a value of zero is regarded as `False`, and a non-zero value is regarded as `True`. For example:

```
bool(False)
print(bool(False))

va1 = 0
print(bool(va1))

va2 = 11
print(bool(va2))

va3 = -2.3
print(bool(va3))
```

The output of the above code will be as follows.

```
False
False
True
True
```

Sequence data types are also a very basic and common data type, which we'll look at next.

Sequences

Sequence data types are used to store multiple values in a single variable in an organized and efficient way. There are four basic sequence types: string, range, lists, and tuples.

Strings

A string is an immutable sequence of characters represented in single, double, or triple quotes.

Immutable means that once a data type has been assigned some value, it can't be changed.

The string type in Python is called `str`. A triple quote string can span into multiple lines that include all the whitespace in the string. For example:

```
str1 = 'Hello how are you'
str2 = "Hello how are you"
str3 = """multiline
       String"""
print(str1)
print(str2)
print(str3)
```

The output will be as follows:

```
Hello how are you
Hello how are you
multiline
String
```

The + operator concatenates strings, which returns a string after concatenating the operands, joining them together. For example:

```
f = 'data'
s = 'structure'

print(f + s)
print('Data ' + 'structure')
```

The output will be as follows:

```
datastructure
Data structure
```

The * operator can be used to create multiple copies of a string. When it is applied with an integer (*n*, let's say) and a string, the * operator returns a string consisting of *n* concatenated copies of the string. For example:

```
st = 'data.'

print(st * 3)
print(3 * st)
```

The output will be as follows.

```
data.data.data.
data.data.data.
```

Range

The `range` data type represents an immutable sequence of numbers. It is mainly used in `for` and `while` loops. It returns a sequence of numbers starting from a given number up to a number specified by the function argument. It is used as in the following command:

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

Here, the `start` argument specifies the start of the sequence, the `stop` argument specifies the end limit of the sequence, and the `step` argument specifies how the sequence should increase or decrease. This example Python code demonstrates the working of the range function:

```
print(list(range(10)))
print(range(10))
print(list(range(10)))
print(range(1,10,2))
print(list(range(1,10,2)))
print(list(range(20,10,-2)))
```

The output will be as follows.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(1, 10, 2)
[1, 3, 5, 7, 9]
[20, 18, 16, 14, 12]
```

Lists

Python lists are used to store multiple items in a single variable. Duplicate values are allowed in a list, and elements can be of different types: for example, you can have both numeric and string data in a Python list.

The items stored in the list are enclosed within square brackets, [], and separated with a comma, as shown below:

```
a = ['food', 'bus', 'apple', 'queen']
print(a)
mylist  = [10, "India", "world", 8]
# accessing elements in list.
print(mylist[1])
```

The output of the above code will be as follows.

```
['food', 'bus', 'apple', 'queen']
India
```

The data element of the list is shown in *Figure 1.4*, showing the index value of each of the list items:

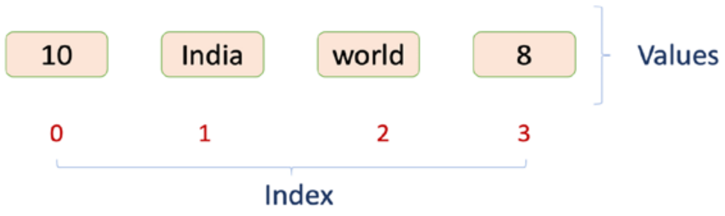


Figure 1.4: Data elements of a sample list

The characteristics of a list in Python are as follows. Firstly, the list elements can be accessed by its index, as shown in *Figure 1.4*. The list elements are ordered and dynamic. It can contain any arbitrary objects that are so desired. In addition, the `list` data structure is mutable, whereas most of the other data types, such as `integer` and `float` are immutable.

Seeing as a list is a mutable data type, once created, the list elements can be added, deleted, shifted, and moved within the list.

All the properties of lists are explained in *Table 1.1* below for greater clarity:

Property	**Description**	**Example**

Ordered	The list elements are ordered in a sequence in which they are specified in the list at the time of defining them. This order does not need to change and remains innate for its lifetime.	`[10, 12, 31, 14] == [14, 10, 31, 12]`
		
		`False`

Dynamic	The list is dynamic. It can grow or shrink as needed by adding or removing list items.	`b = ['data', 'and', 'book', 'structure', 'hello', 'st']`
		`b += [32]`
		`print(b)`
		`b[2:3] = []`
		`print(b)`
		`del b[0]`
		`print(b)`
		
		`['data', 'and', 'book', 'structure', 'hello', 'st', 32]`
		`['data', 'and', 'structure', 'hello', 'st', 32]`
		`['and', 'structure', 'hello', 'st', 32]`

List elements can be any arbitrary set of objects	List elements can be of the same type or varying data types.	`a = [2.2, 'python', 31, 14, 'data', False, 33.59]`
		`print(a)`
		
		`[2.2, 'python', 31, 14, 'data', False, 33.59]`

List elements can be accessed through an index	Elements can be accessed using zero-based indexing in square brackets, similar to a string. Accessing elements in a list is similar to strings; negative list indexing also works in lists. A negative list index counts from the end of the list.
	
	Lists also support slicing. If `abc` is a list, the expression `abc[x:y]` will return the portion of elements from index `x` to index `y` (not including index `y`)
		`a = ['data', 'structures', 'using', 'python', 'happy', 'learning']`
		`print(a[0])`
		`print(a[2])`
		`print(a[-1])`
		`print(a[-5])`
		`print(a[1:5])`
		`print(a[-3:-1])`
		
		`data`
		`using`
		`learning`
		`structures`
		`['structures', 'using', 'python', 'happy']`
		`['python', 'happy']`

Mutable	Single list value: Elements in a list can be updated through indexing and simple assignment.
	
	Modifying multiple list values is also possible through slicing.
		`a = ['data', 'and', 'book', 'structure', 'hello', 'st']`
		`print(a)`
		`a[1] = 1`
		`a[-1] = 120`
		`print(a)`
		`a = ['data', 'and', 'book', 'structure', 'hello', 'st']`
		`print(a[2:5])`
		`a[2:5] = [1, 2, 3, 4, 5]`
		`print(a)`

		`['data', 'and', 'book',`
		`'structure', 'hello', 'st']`
		`['data', 1, 'book',`
		`'structure', 'hello', 120]`
		`['book', 'structure',`
		`'hello']`
		`['data', 'and', 1, 2, 3, 4,`
		`5, 'st']`

Other operators	Several operators and built-in functions can also be applied in lists, such as `in`, `not in`, concatenation (+), and replication (*) operators. Moreover, other built-in functions, such as `len()`, `min()`, and `max()`, are also available.	`a = ['data', 'structures',`
		`'using', 'python', 'happy',`
		`'learning']`
		`print('data' in a)`
		`print(a)`
		`print(a + ['New', 'elements'])`
		`print(a)`
		`print(a *2)`
		`print(len(a))`
		`print(min(a))`
		
		`['data', 'structures',`
		`'using', 'python', 'happy',`
		`'learning']`
		`['data', 'structures',`
		`'using', 'python', 'happy',`
		`'learning', 'New',`
		`'elements']`
		`['data', 'structures',`
		`'using', 'python', 'happy',`
		`'learning']`
		`['data', 'structures',`
		`'using', 'python',`
		`'happy', 'learning',`
		`'data', 'structures',`
		`'using', 'python', 'happy',`
		`'learning']`
		`6`
		`data`

Table 1.1: Characteristics of list data structures with examples

Now, while discussing list data types, we should first understand different operators, such as membership, identity, and logical operators, before discussing them and how they can be used in list data types or any other data types. In the coming section, we discuss how these operators work and are used in various data types.

Membership, identity, and logical operations

Python supports membership, identity, and logical operators. Several data types in Python support them. In order to understand how these operators work, we'll discuss each of these operations in this section.

Membership operators

These operators are used to validate the membership of an item. Membership means we wish to test if a given value is stored in the sequence variable, such as a string, list, or tuple. Membership operators are to test for membership in a sequence; that is, a string, list, or tuple. Two common membership operators used in Python are `in` and `not in`.

The `in` operator is used to check whether a value exists in a sequence. It returns `True` if it finds the given variable in the specified sequence, and `False` if it does not:

```
# Python program to check if an item (say second
# item in the below example) of a list is present
# in another list (or not) using 'in' operator
mylist1 = [100,20,30,40]
mylist2 = [10,50,60,90]
if mylist1[1] in mylist2:
    print("elements are overlapping")
else:
    print("elements are not overlapping")
```

The output will be as follows:

```
elements are not overlapping
```

The '`not in`' operator returns to `True` if it does not find a variable in the specified sequence and `False` if it is found:

```
val = 104
mylist = [100, 210, 430, 840, 108]
if val not in mylist:
    print("Value is NOT present in mylist")
```

```
else:
    print("Value is  present in mylist")
```

The output will be as follows.

```
Value is NOT present in mylist
```

Identity operators

Identity operators are used to compare objects. The different types of identity operators are `is` and `is not`, which are defined as follows.

The `is` operator is used to check whether two variables refer to the same object. This is different from the equality (==) operator. In the equality operator, we check whether two variables are equal. It returns `True` if both side variables point to the same object; if not, then it returns `False`:

```
Firstlist = []
Secondlist = []
if Firstlist == Secondlist:
    print("Both are equal")
else:
    print("Both are not equal")

if Firstlist is Secondlist:
    print("Both variables are pointing to the same object")
else:
    print("Both variables are not pointing to the same object")

thirdList = Firstlist

if thirdList is Secondlist:
    print("Both are pointing to the same object")
else:
    print("Both are not pointing to the same object")
```

The output will be as follows:

```
Both are equal
Both variables are not pointing to the same object
Both are not pointing to the same object
```

The `is not` operator is used to check whether two variables point to the same object or not. `True` is returned if both side variables point to different objects, otherwise, it returns `False`:

```
Firstlist = []
Secondlist = []
if Firstlist is not Secondlist:
  print("Both Firstlist and Secondlist variables are the same object")
else:
  print("Both Firstlist and Secondlist variables are not the same object")
```

The output will be as follows:

```
Both Firstlist and Secondlist variables are not the same object
```

This section was about identity operators. Next, let us discuss logical operators.

Logical operators

These operators are used to combine conditional statements (`True` or `False`). There are three types of logical operators: `AND`, `OR`, and `NOT`.

The logical `AND` operator returns True if both the statements are true, otherwise it returns False. It uses the following syntax: A and B:

```
a = 32
b = 132
if a > 0 and b > 0:
  print("Both a and b are greater than zero")
else:
  print("At least one variable is less than 0")
```

The output will be as follows.

```
Both a and b are greater than zero
```

The logical `OR` operator returns True if any of the statements are true, otherwise it returns False. It uses the following syntax: A or B:

```
a = 32
b = -32
if a > 0 or b > 0:
  print("At least one variable is greater than zero")
```

```
else:
  print("Both variables are less than 0")
```

The output will be as follows.

```
At least one variable is greater than zero
```

The logical `NOT` operator is a Boolean operator, which can be applied to any object. It returns `True` if the object/operand is false, otherwise it returns `False`. Here, the operand is the unary expression/statement on which the operator is applied. It uses the following syntax: `not A`:

```
a = 32
if not a:
  print("Boolean value of a is False")
else:
  print("Boolean value of a is True")
```

The output will be as follows.

```
Boolean value of a is True
```

In this section, we learned about different operators available in Python, and also saw how membership and identity operators can be applied to list data types. In the next section, we will continue discussing a final sequence data type: tuples.

Tuples

Tuples are used to store multiple items in a single variable. It is a read-only collection where data is ordered (zero-based indexing) and unchangeable/immutable (items cannot be added, modified, removed). Duplicate values are allowed in a tuple, and elements can be of different types, similar to lists. Tuples are used instead of lists when we wish to store the data that should not be changed in the program.

Tuples are written with round brackets and items are separated by a comma:

```
tuple_name = ("entry1", "entry2", "entry3")
```

For example:

```
my_tuple = ("Shyam", 23, True, "male")
```

Tuples support + (concatenation) and * (repetition) operations, similar to strings in Python. In addition, a membership operator and iteration operation are also available in a tuple. Different operations that tuples support are listed in *Table 1.2:*

Expression	Result	Description
`print(len((4,5, "hello")))`	`3`	Length
`print((4,5)+(10,20))`	`(4,5,10,20)`	Concatenation
`print((2,1)*3)`	`(2,1,2,1,2,1)`	Repetition
`print(3 in ('hi', 'xyz',3))`	`True`	Membership

`for p in (6,7,8):`
`    print(p)`
	`6,7,8`	Iteration

Table 1.2: Example of tuple operations

Tuples in Python support zero-based indexing, negative indexing, and slicing. To understand it, let's take a sample tuple, as shown below:

```
x = ( "hello", "world", " india")
```

We can see examples of zero-based indexing, negative indexing, and slicing operations in *Table 1.3*:

Expression	Result	Description
`print(x[1])`	`"world"`	Zero-based indexing means that indexing starts from 0 rather than 1, and hence in this example, the first index refers to the second member of the tuple.
`print(x[-2])`	`"world"`	Negative: counting from the right-hand side.
`print(x[1:])`	`("world", "india")`	Slicing fetches a section.

Table 1.3: Example of tuple indexing and slicing

Complex data types

We have discussed basic data types. Next, we discuss complex data types, which are mapping data types, in other words, dictionary, and set data types, namely, set and frozenset. We will discuss these data types in detail in this section.

Dictionaries

In Python, a dictionary is another of the important data types, similar to a list, in the sense that it is also a collection of objects. It stores the data in unordered {key-value} pairs; a key must be of a hashable and immutable data type, and value can be any arbitrary Python object. In this context, an object is hashable if it has a hash value that does not change during its lifetime in the program.

Items in the dictionary are enclosed in curly braces, `{}`, separated by a comma, and can be created using the `{key:value}` syntax, as shown below:

```
dict = {
    <key>: <value>,
    <key>: <value>,
      .
      .
      .
    <key>: <value>
}
```

Keys in dictionaries are case-sensitive, they should be unique, and cannot be duplicated; however, the values in the dictionary can be duplicated. For example, the following code can be used to create a dictionary:

```
my_dict = {'1': 'data',
           '2': 'structure',
           '3': 'python',
           '4': 'programming',
           '5': 'language'
          }
```

Figure 1.5 shows the `{key-value}` pairs created by the preceding piece of code:

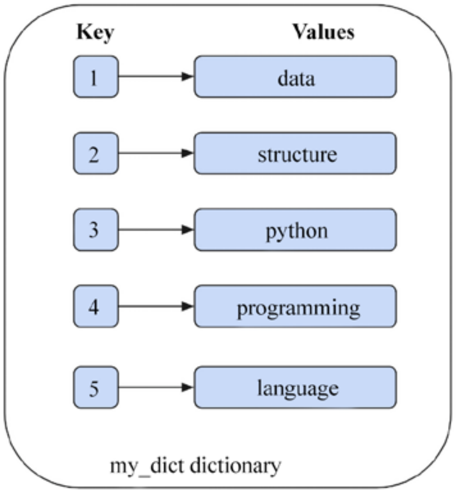


Figure 1.5: Example dictionary data structure

Values in a dictionary can be fetched based on the key. For example: `my_dict['1']` gives `data` as the output.

The `dictionary` data type is mutable and dynamic. It differs from lists in the sense that dictionary elements can be accessed using keys, whereas the list elements are accessed via indexing. *Table 1.4* shows different characteristics of the dictionary data structure with examples:

Item	**Example**
Creating a dictionary, and accessing elements from a dictionary	`person = {}` `print(type(person))` `person['name'] = 'ABC'` `person['lastname'] = 'XYZ'` `person['age'] = 31` `person['address'] = ['Jaipur']` `print(person)` `print(person['name'])` Output: `<class 'dict'>{'name': 'ABC', 'lastname': 'XYZ', 'age': 31, 'address': ['Jaipur']}ABC`
`in` and `not in` operators	`print('name' in person)` `print('fname' not in person)` Output: `True` `True`
Length of the dictionary	`print(len(person))` Output: `4`

Table 1.4: Characteristics of dictionary data structures with examples

Python also includes the dictionary methods as shown in *Table 1.5*:

Function	Description	Example

`mydict.clear()`	Removes all elements from a dictionary.	`mydict = {'a': 1, 'b': 2, 'c': 3}`
		`print(mydict)`
		`mydict.clear()`
		`print(mydict)`
		
		`{'a': 1, 'b': 2, 'c': 3}`
		`{}`

`mydict.get(<key>)`	Searches the dictionary for a key and returns the corresponding value, if it is found; otherwise, it returns `None`.	`mydict = {'a': 1, 'b': 2, 'c': 3}`
		`print(mydict.get('b'))`
		`print(mydict)`
		`print(mydict.get('z'))`
		
		`2`
		`{'a': 1, 'b': 2, 'c': 3}`
		`None`

`mydict.items()`	Returns a list of dictionary items in (key, value) pairs.	`print(list(mydict.items()))`
		
		`[('a', 1), ('b', 2), ('c', 3)]`

`mydict.keys()`	Returns a list of dictionary keys.	`print(list(mydict.keys()))`
		
		`['a', 'b', 'c']`

`mydict.values()`	Returns a list of dictionary values.	`print(list(mydict.values()))`
		
		`[1, 2, 3]`

`mydict.pop()`	If a given key is present in the dictionary, then this function will remove the key and return the associated value.	`print(mydict.pop('b'))`
		`print(mydict)`
		
		`{'a': 1, 'c': 3}`

`mydict.popitem()`	This method removes the last key-value pair added in the dictionary and returns it as a tuple.	`mydict = {'a': 1,'b': 2,'c': 3}`
		`print(mydict.popitem())`
		`print(mydict)`
		
		`{'a': 1, 'b': 2}`

`mydict.update(<obj>)`	Merges one dictionary with another. Firstly, it checks whether a key of the second dictionary is present in the first dictionary; the corresponding value is then updated. If the key is not present in the first dictionary, then the key-value pair is added.	`d1 = {'a': 10, 'b': 20, 'c': 30}`
		`d2 = {'b': 200, 'd': 400}`
		`print(d1.update(d2))`
		`print(d1)`
		
		`{'a': 10, 'b': 200, 'c': 30, 'd': 400}`

Table 1.5: List of methods of dictionary data structures

Sets

A set is an unordered collection of hashable objects. It is iterable, mutable, and has unique elements. The order of the elements is also not defined. While the addition and removal of items are allowed, the items themselves within the set must be immutable and hashable. Sets support membership testing operators (`in, not in`), and operations such as intersection, union, difference, and symmetric difference. Sets cannot contain duplicate items. They are created by using the built-in `set()` function or curly braces `{}`. A `set()` returns a set object from an iterable. For example:

```
x1 = set(['and', 'python', 'data', 'structure'])
print(x1)
print(type(x1))
x2 = {'and', 'python', 'data', 'structure'}
print(x2)
```

The output will be as follows:

```
{'python', 'structure', 'data', 'and'}
<class 'set'>
{'python', 'structure', 'data', 'and'}
```

It is important to note that sets are unordered data structures, and the order of items in sets is not preserved. Therefore, your outputs in this section may be slightly different than those displayed here. However, this does not affect the function of the operations we will be demonstrating in this section.

Sets are generally used to perform mathematical operations, such as intersection, union, difference, and complement. The `len()` method gives the number of items in a set, and the `in` and `not in` operators can be used in sets to test for membership:

```
x = {'data', 'structure', 'and', 'python'}
print(len(x))
print('structure' in x)
```

The output will be as follows:

```
4
True
```

The most commonly used methods and operations that can be applied to `set` data structures are as follows. The union of the two sets, say, `x1` and `x2`, is a set that consists of all elements in either set:

```
x1 = {'data', 'structure'}
x2 = {'python', 'java', 'c', 'data'}
```

Figure 1.6 shows a Venn diagram demonstrating the relationship between the two sets:

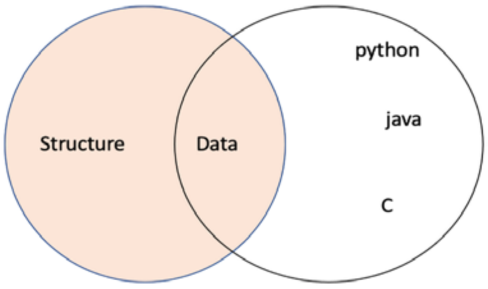


Figure 1.6: Venn diagram of sets

A description of the various operations that can be applied on set type variables is shown, with examples, in *Table 1.6*:

Description	**Example sample code**

Union of two sets, `x1` and `x2`. It can be done using two methods, (1) using the | operator, (2) using the `union` method.	`x1 = {'data', 'structure'}`
	`x2 = {'python', 'java', 'c', 'data'}`
	`x3 = x1 | x2`
	`print(x3)`
	`print(x1.union(x2))`
	
	`{'structure', 'data', 'java', 'c', 'python'}`
	`{'structure', 'data', 'java', 'c', 'python'}`

Intersection of sets: to compute the intersection of two sets, an & operator and the `intersection()` method can be used, which returns a set of items common to both sets, `x1` and `x2`.	`print(x1.intersection(x2))`
	`print(x1 & x2)`
	
	`{'data'}`
	`{'data'}`

The difference between sets can be obtained using `.difference()` and the subtraction operator, `-`, which returns a set of all elements that are in `x1`, but not in `x2`.	`print(x1.difference(x2))`
	`print(x1 - x2)`
	
	`{'structure'}`
	`{'structure'}`

Symmetric difference can be obtained using `.symmetric_difference()`, while ^ returns a set of all data items that are present in either `x1` or `x2`, but not both.	`print(x1.symmetric_difference(x2))`
	`print(x1 ^ x2)`
	
	`{'structure', 'python', 'c', 'java'}`
	`{'structure', 'python', 'c', 'java'}`

To test whether a set is a subset of another, use `.issubset()` and the operator <=.	`print(x1.issubset(x2))`
	`print(x1 <= x2)`
	
	`False`
	`False`

Table 1.6: Description of various operations applicable to set type variables

Immutable sets

In Python, `frozenset` is another built-in type data structure, which is, in all respects, exactly like a set, except that it is immutable, and so cannot be changed after creation. The order of the elements is also undefined. A `frozenset` is created by using the built-in function `frozenset()`:

```
x = frozenset(['data', 'structure', 'and', 'python'])
print(x)
```

The output is:

```
frozenset({'python', 'structure', 'data', 'and'})
```

Frozensets are useful when we want to use a set but require the use of an immutable object. Moreover, it is not possible to use set elements in the set, since they must also be immutable. Consider an example:

```
a11 = set(['data'])
a21 = set(['structure'])
a31 = set(['python'])
x1 = {a11, a21, a31}
```

The output will be:

```
TypeError: unhashable type: 'set'
```

Now with frozenset:

```
a1 = frozenset(['data'])
a2 = frozenset(['structure'])
a3 = frozenset(['python'])
x = {a1, a2, a3}
print(x)
```

The output is:

```
{frozenset({'structure'}), frozenset({'python'}), frozenset({'data'})}
```

In the above example, we create a set `x` of frozensets (`a1`, `a2`, and `a3`), which is possible because the frozensets are immutable.

We have discussed the most important and popular data types available in Python. Python also provides a collection of other important methods and modules, which we will discuss in the next section.

Python's collections module

The `collections` module provides different types of containers, which are objects that are used to store different objects and provide a way to access them. Before accessing these, let's consider briefly the role and relationships between modules, packages, and scripts.

A module is a Python script with the `.py` extension that contains a collection of functions, classes, and variables. A package is a directory that contains collections of modules; it has an `__init__.py` file, which lets the interpreter know that it is a package. A module can be called into a Python script, which can in turn make use of the module's functions and variables in its code. In Python, we can import these to a script using the `import` statement. Whenever the interpreter encounters the `import` statement, it imports the code of the specified module.

Table 1.7 provides the data types and operations of the collections module and their descriptions:

Container data type	**Description**
`namedtuple`	Creates a `tuple` with named fields similar to regular tuples.
`deque`	Doubly-linked lists that provide efficient adding and removing of items from both ends of the list.
`defaultdict`	A `dictionary` subclass that returns default values for missing keys.
`ChainMap`	A `dictionary` that merges multiple dictionaries.
`Counter`	A `dictionary` that returns the counts corresponding to their objects/key.
`UserDict UserList UserString`	These data types are used to add more functionalities to their base data structure, such as a `dictionary`, `list`, and `string`. And we can create subclasses from them for custom `dict`/`list`/`string`.

Table 1.7: Different container data type of the collections module

Let's consider these types in more detail.

Named tuples

The `namedtuple` of `collections` provides an extension of the built-in tuple data type. `namedtuple` objects are immutable, similar to standard tuples. Thus, we can't add new fields or modify existing ones after the `namedtuple` instance is created. They contain keys that are mapped to a particular value, and we can iterate through named tuples either by index or key. The `namedtuple` function is mainly useful when several tuples are used in an application and it is important to keep track of each of the tuples in terms of what they represent.

In this situation, namedtuple presents a more readable and self-documenting method. The syntax is as follows:

```
nt = namedtuple(typename , field_names)
```

Here is an example:

```
from collections import namedtuple
Book = namedtuple ('Book', ['name', 'ISBN', 'quantity'])
Book1 = Book('Hands on Data Structures', '9781788995573', '50')
#Accessing data items
print('Using index ISBN:' + Book1[1])
print('Using key ISBN:' + Book1.ISBN)
```

The output will be as follows.

```
Using index ISBN:9781788995573
Using key ISBN:9781788995573
```

Here, in the above code, we firstly imported `namedtuple` from the `collections` module. `Book` is a named tuples, "`class`," and then, `Book1` is created, which is an instance of `Book`. We also see that the data elements can be accessed using index and key methods.

Deque

A deque is a double-ended queue (deque) that supports append and pop elements from both sides of the list. Deques are implemented as double-linked lists, which are very efficient for inserting and deleting elements in O(1) time complexity.

Consider an example:

```
from collections import deque
s = deque()   # Creates an empty deque
print(s)
my_queue = deque([1, 2, 'Name'])
print(my_queue)
```

The output will be as follows.

```
deque([])
deque([1, 2, 'Name'])
```

You can also use some of the following predefined functions:

Function	Description
`my_queue.append('age')`	Insert `'age'` at the right end of the list.
`my_queue.appendleft('age')`	Insert `'age'` at the left end of the list.
`my_queue.pop()`	Delete the rightmost value.
`my_queue.popleft()`	Delete the leftmost value.

Table 1.8: Description of different queue functions

In this section, we showed the use of the `deque` method of the `collections` module, and how elements can be added and deleted from the queue.

Ordered dictionaries

An ordered dictionary is a dictionary that preserves the order of the keys that are inserted. If the key order is important for any application, then `OrderedDict` can be used:

```
od = OrderedDict([items])
```

An example could look like the following:

```
from collections import OrderedDict
od = OrderedDict({'my': 2, 'name ': 4, 'is': 2, 'Mohan' :5})
od['hello'] = 4
print(od)
```

The output will be as follows.

```
OrderedDict([('my', 2), ('name ', 4), ('is', 2), ('Mohan', 5), ('hello',
4)])
```

In the above code, we create a dictionary, `od`, using the `OrderedDict` module. We can observe that the order of the keys is the same as the order when we created the key.

Default dictionary

The default dictionary (`defaultdict`) is a subclass of the built-in dictionary class (`dict`) that has the same methods and operations as that of the `dictionary` class, with the only difference being that it never raises a `KeyError`, as a normal dictionary would. `defaultdict` is a convenient way to initialize dictionaries:

```
d = defaultdict(def_value)
```

An example could look like the following:

```
from collections import defaultdict
dd = defaultdict(int)
words = str.split('data python data data structure data python')
for word in words:
    dd[word] += 1
print(dd)
```

The output will be as follows.

```
defaultdict(<class 'int'>, {'data': 4, 'python': 2, 'structure': 1})
```

In the above example, if an ordinary dictionary had been used, then Python would have shown `KeyError` while the first key was added. `int`, which we supplied as an argument to `defaultdict`, is really the `int()` function, which simply returns a zero.

ChainMap object

`ChainMap` is used to create a list of dictionaries. The `collections.ChainMap` data structure combines several dictionaries into a single mapping. Whenever a key is searched in the `chainmap`, it looks through all the dictionaries one by one, until the key is not found:

```
class collections.ChainMap(dict1, dict2)
```

An example could look like the following:

```
from collections import ChainMap
dict1 = {"data": 1, "structure": 2}
dict2 = {"python": 3, "language": 4}
chain = ChainMap(dict1, dict2)
print(chain)
print(list(chain.keys()))
print(list(chain.values()))
print(chain["data"])
print(chain["language"])
```

The output will be:

```
ChainMap({'data': 1, 'structure': 2}, {'python': 3, 'language': 4})
['python', 'language', 'data', 'structure']
[3, 4, 1, 2]
```

```
1
4
```

In the above code, we create two dictionaries, namely, `dict1` and `dict2`, and then we can combine both of these dictionaries using the `ChainMap` method.

Counter objects

As we discussed earlier, a hashable object is one whose hash value will remain the same during its lifetime in the program. `counter` is used to count the number of hashable objects. Here, the dictionary key is a hashable object, while the corresponding value is the count of that object. In other words, `counter` objects create a hash table in which the elements and their count are stored as dictionary keys and value pairs.

`Dictionary` and `counter` objects are similar in the sense that data is stored in a `{key, value}` pair, but in `counter` objects, the value is the count of the key whereas it can be anything in the case of `dictionary`. Thus, when we only want to see how many times each unique word is occurring in a string, we use the `counter` object.

An example could look like the following:

```
from collections import Counter
inventory = Counter('hello')
print(inventory)
print(inventory['l'])
print(inventory['e'])
print(inventory['o'])
```

The output will be:

```
Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1})
2
1
1
```

In the above code, the `inventory` variable is created, which holds the counts of all the characters using the `counter` module. The count values of these characters can be accessed using dictionary-like key access (`[key]`).

UserDict

Python supports a container, `UserDict`, present in the collections module, that wraps the dictionary objects. We can add customized functions to the dictionary. This is very useful for applications where we want to add/update/modify the functionalities of the dictionary. Consider the example code below where pushing/adding a new data element is not allowed in the dictionary:

```python
# we can not push to this user dictionary
from collections import UserDict
class MyDict(UserDict):
    def push(self, key, value):
        raise RuntimeError("Cannot insert")
d = MyDict({'ab':1, 'bc': 2, 'cd': 3})
d.push('b', 2)
```

The output is as follows:

```
RuntimeError: Cannot insert
```

In the above code, a customized push function in the `MyDict` class is created to add the customized functionality, which does not allow you to insert an element into the dictionary.

UserList

A `UserList` is a container that wraps list objects. It can be used to extend the functionality of the `list` data structure. Consider the example code below, where pushing/adding a new data element is not allowed in the `list` data structure:

```python
# we can not push to this user list
from collections import UserList
class MyList(UserList):
    def push(self, key):
        raise RuntimeError("Cannot insert in the list")
d = MyList([11, 12, 13])
d.push(2)
```

The output is as follows:

```
RuntimeError: Cannot insert in the list
```

In the above code, a customized `push` function in the `MyList` class is created to add the functionality to not allow you to insert an element into the `list` variable.

UserString

Strings can be considered as an array of characters. In Python, a character is a string of one length and acts as a container that wraps a string object. It can be used to create strings with customized functionalities. An example could look like the following:

```
#Create a custom append function for string
from collections import UserString
class MyString(UserString):
    def append(self, value):
        self.data += value
s1 = MyString("data")
print("Original:", s1)
s1.append('h')
print("After append: ", s1)
```

The output is:

```
Original: data
After append:  datah
```

In the above example code, a customized append function in the `MyString` class is created to add the functionality to append a string.

Summary

In this chapter, we have discussed different built-in data types supported by Python. We have also looked at a few basic Python functions, libraries, and modules, such as the collections module. The main objective of this chapter was to give an overview of Python and make a user acquainted with the language so that it is easy to implement the advanced algorithms of data structures.

Overall, this chapter has provided an overview of several data structures available in Python that are pivotal for understanding the internals of data structures. In the next chapter, we will introduce the basic concepts of algorithm design and analysis.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://packt.link/MEvK4`