Python Programming for Everyone

Learn to write programs more effectively in no time

JUBRAN AKRAM

A tutorial $\boxed{\mathcal{AT}}$

Contents

Chapter 1

Introduction

Hello Everyone, Welcome to this tutorial!

This tutorial is made from my **Python Programming for Everyone** video series which is available on YouTube link. It is not only suitabe for a complete beginner who wants to learn python programming, but also for experienced programmers. Using many examples, it explains the basics as well as some useful tricks to write python codes more effectively.

1.1 Programming language

> **A programming language** is a set of grammatical rules for instructing a computer to perform specific tasks.

In the above definition, the important thing to note is the set of rules and structure of instructions that we need to follow when writing programs in any specific language, and we'll cover some of that for python in this tutorial.

There exists a wide variety of programming languages. We can find a comprehensive list of programming languages in this wikipedia link.

Question: Why choose python programming then?

Figure 1.1 shows the results of a web search on the top 10 programming languages that are more desirable to learn in 2020. Python is one of the top 3 programming languages in all of the six lists obtained from different websites. It is used in web development, game development, machine learning, data science, data visualization and in many more applications.

1.1.1 Python programming

Python was developed by Guido van Rossum and was released in 1991.

> **Python** is an **interpreted**, **high-level**, and **general-purpose** programming language

Let's see what are these terms (interpreted, high-level, and general-purpose).

Figure 1.1: Web search results on top 10 programming languages that are more desirable to learn in 2020.

Interpreted programming language

The program that we write is called the **source code**. For machines to understand this code, it needs translations which is done using compilers and interpreters.

An interpreted language is "interpreted" in their original source code or merely compiled in real time when it's run. An interpreter translates the program one statement at a time i.e. first line is evaluated first and then the second and so on. By doing so, source code can be analyzed efficiently as any error will be known immediately and debugging is relatively easier. However, all this process can add to the execution time.

High-level programming language

There are two main types of programming languages 1. low level 2. high level

Figure 1.2 explains different types and levels of programming languages. The lowest level programming language is the machine code which is binary code read by hardware but not by humans. The first layer of human readable code is the assembly language. This is followed by middle-level, high-level and interpeted languages.

Unlike low-level programming languages, high-level programming languages are easier to understand for humans (programmer friendly) as instead of binary code, everything is written in plain english language. Consequently, the codes written in a high-level programming language are also easier to debug and maintain. These are also portable and less memory efficient as compared to the low-level programming languages. The program execution speed in a high-level programming language is relatively lower as compared to the low-level programming languages.

General-purpose programming language

> **A general-purpose programming language** is designed to be used for writing software in the widest variety of application domains (Wikipedia).

It is not domain specific such as database query languages.

Python supports cross-platform operating systems and it is used in wide variety of applications such as web development, game development, machine learning, data science, data visualization etc.

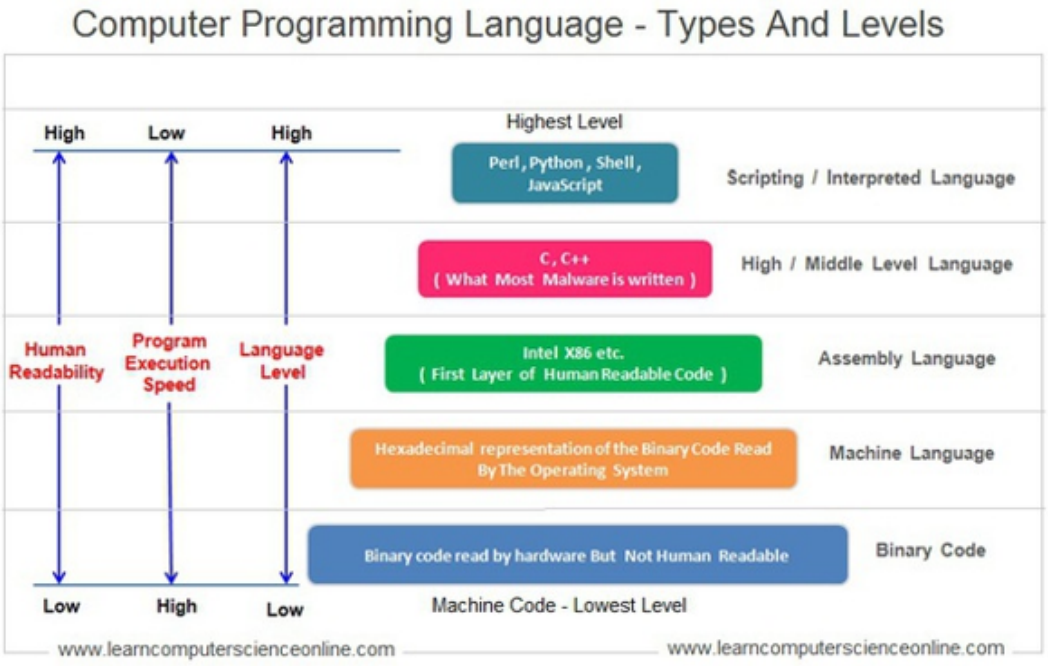


Figure 1.2: Programming languages types and levels (source: www.learncomputerscienceonline.com)

1.2 Installing python

There are two easy ways to get python installed on your machine:

1. Go to `https://www.python.org/downloads/`, download the right python file for your operating system and run it on your machine. It is as simple as that.
2. Install the anaconda python distribution. Go to `https://docs.anaconda.com/anaconda/install/` and follow the instructions to install python on windows, macOS, or Linux. Anaconda distribution comes with 1,500+ packages selected from PyPI as well as the conda package and virtual environment managers.

Additional packages can be installed using either pip or conda package managers. The main difference between the two package managers is in how the package dependencies are managed. For example, we can use one of the following to install new packages

```
pip install package_name
```

```
conda install package_name
```

1.3 Code editor and Interactive development environment (IDE)

A code editor is a light-weight tool for writing and editing codes. Although we can use a simple notepad utility for writing programs, a code editor offers a bit-refined option as it is optimized for different programming languages. Some examples of code editors include notepad++, sublime text and atom.

An integrated development environment (IDE) provides comprehensive facilities to programmers for software development. It typically includes a source code editor, build automation tools and

a debugger to write programs efficiently. Some examples of the IDEs include Spyder, Eclipse, PyCharm and Visual Studio Code.

You can choose to work either in a code editor or an IDE and will still be able to follow the examples.

Summary

In this chapter, we learned that python is an interpreted, high-level and general-purpose programming language. We then discussed what are the interpreted, high-level and general-purpose programming languages. We then described different ways of installing python on our machines. Finally, we explained the difference between a code editor and an IDE.

Chapter 2

Comments and print()

2.1 Comments

As programs get bigger and more complicated, they become more difficult to read. **Comments** are very useful for organizing and explaining the programs and also for keeping notes for yourself and for others to follow your program easily, thus improving the code readibility.

There are two types of comments: 1. single-line comments 2. multi-line comments

2.1.1 Single-line comments

> **A single-line comment** starts with the hash or pound character (#) followed by the comment. It does not automatically continue to the next line. If we want to continue the comment on the next line, we will start the next line with another # character.

These comments have no impact on the output of any program. The python interpreter simply ignores any statement after the hash key. We can start a single-line comment by typing the hash or pound character followed by the comment -

These comments have no impact on the output of any program. The python interpreter simply ignores any statement after the hash key.

[1]:
```
# This is a single-line comment
```

We can also start a single-line comment on the right of the code written on a given line.

[2]:
```
num = 2         # This is an arbitrary number
```

2.1.2 Multi-line comments

Unlike other programming languages, there's no feature as multi-line comment in Python. However, # single-line comments can be used on consecutive lines to mimic a multi-line comment.

[3]:
```
# This is a multi-line comment
# write as many lines as you want to ...
```

We can also write multiline comments in triple quotes. These are similar to multi-line strings but can be used as comments in python programming. The Style Guide for Python Code states that each script should start with a docstring that explains the script's purpose.

[5]:
```
""" This is a
multi-line comment. you can write as
many lines as you want to ...."""
```

[5]:
```
' This is a \nmulti-line comment. you can write as\nmany lines as you want to
...'
```

2.2 print()

The **print()** function allows showing messages or results on the screen or another standard output device.

Let's look at a simple example, in which we wish to print "Hello World!" on the screen. To do this, we write "Hello World" in quotes as the argument of the print function. The arguments are written inside the parenthesis of a function.

[6]:
```
print('Hello, World!')
```

```
Hello, World!
```

We can also write "Hello World" as a string variable, let's say greeting, and use it as an argument of the print function.

[7]:
```
greeting = 'Hello, World!'
print(greeting)
```

```
Hello, World!
```

We can also use any number as the argument

[8]:
```
num = 2.5
print(num)
```

```
2.5
```

Now let's say we want to print multiple texts at the same time, for example, "Hello" and "How are you?". The print() function will show both items together with a space as the separator.

[9]:
```
print('Hello', 'how are you?')
```

```
Hello how are you?
```

We can customize the separator. Instead of a space, we can include a comma or a new line by setting the **sep** parameter to ',' and '\n' respectively.

[10]:
```
print('Hello', 'how are you?', sep=', ')
print('Hello', 'how are you?', sep='\n')
```

```
Hello, how are you?
Hello
how are you?
```

Now let's say, we want to print a name (that we can change dynamically) in the text, for example, "Hello Jubran, how are you?"

We can write the desired print statement in many ways.

[12]:
```
name = "Jubran"
print('Hello '+name+', how are you?')            # string concatenation
print('Hello %s, how are you?' %name)            # format specifier
print('Hello {}, how are you?'.format(name))     # .format method
print(f'Hello {name}, how are you?')             # formatted strings (f-strings)
```

```
Hello Jubran, how are you?
Hello Jubran, how are you?
Hello Jubran, how are you?
Hello Jubran, how are you?
```

Let's see another example, in which we wish to include a string variable and an integer in the print statement.

[14]:
```
print('John has 5 years of work experience in the stock exchange.')
name = 'Steve'
num_years = 10
print(f'{name} has {num_years} years of work experience in the stock exchange.')
```

```
John has 5 years of work experience in the stock exchange.
Steve has 10 years of work experience in the stock exchange.
```

Summary

In this chapter, we learned about how to write single-line and multi-line comments in python. We also learned how to use print() function to show messages or results on the screen effectively.

Chapter 3

Data types/structures

Data types are simply variables that we use to reserve some space in memory. Commonly used data types in python are as follows:

[2]:
```
horizontal_line = 26*"--"
print(f'{10*"--"} Data types/structures {4*"--"}\n')
data_types = ['1. Numeric (integers, floats, complex numbers)',
              '2. Strings', '3. Boolean', '4. Tuples',
              '5. Lists', '6. Sets', '7. Dictionaries']
print(*data_types, sep='\n')
print(horizontal_line)
```

```
-------------------- Data types/structures --------

1. Numeric (integers, floats, complex numbers)
2. Strings
3. Boolean
4. Tuples
5. Lists
6. Sets
7. Dictionaries
--------------------------------------------------
```

We will learn about the details of these data structures in the later chapters. Here, we just want to see, how we can find out the class of an object and its associated properties and methods.

Use **type()** to find out the class of a given object.

Use **dir()** to get a list of methods and properties of a given class.

Use **help()** to find more details on any method and property of a given class.

[3]:
```
data_object = {'a': 1, 'b': 4}
print(f'Data object = {data_object} belongs to {type(data_object)}.',
      horizontal_line, sep='\n')
print(f'The attributes and methods for {type(data_object)} are: \n',
```

```
      dir(data_object), horizontal_line, sep='\n')
print(help(dict.update))
```

```
Data object = {'a': 1, 'b': 4} belongs to <class 'dict'>.
--------------------------------------------------
The attributes and methods for <class 'dict'> are:

['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop',
'popitem', 'setdefault', 'update', 'values']
--------------------------------------------------
Help on method_descriptor:

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]

None
```

Change the data_object in the above example to a tuple, list or set and see their class methods and properties.

Summary

In this chapter, we learned about the basic data types/structures used in python programming. We also learned how to find the class of a given data structure, if unknown. In addition, we learned how to find more details on the properties and methods in a specific object class.

Chapter 4

Strings

A string is a contiguous set of characters represented in the quotation marks.

This definition has three important parts 1. characters 2. contiguous 3. quotation marks

A character is the smallest possible component of a text. This could be any letter, any digit, any punctuation mark or white space (A, B, C, . . . 1, 2, 3, ..., -, . . .).

Contiguous is when different things are placed next to or together in squence (H e l l o)

Finally, this contiguous set of characters is enclosed in quotation marks either single or double quotes to form a string.

4.1 Access substrings

Since a string is basically characters written in sequence, each character has an index, the first character being at index 0. This means that we can access characters from a string using their respective indices.

[4]:
```
str_1 = 'Hello'   # each character can be accessed using its respective indices␣
 ↪i.e. str_1[index]
str_1[0]
```

[4]:
```
'H'
```

[5]:
```
len(str_1)        # len() function outputs the length of a sequence
```

[5]:
```
5
```

We can also use the slicing features to extract substrings

[8]:
```
print(str_1[1:3])     #1 is inclusive whereas 3 is exclusive, thus the desired␣
 ↪indices are [1, 2]
print(str_1[:3])      # upto the third index [0, 1, 2]
print(str_1[3:])      # third index to the last index
```

```
print(str_1[-1])      # last index
print(str_1[::2])     # all values at indices with an interval of 2 [0, 2, ..
 ↪upto the length of the sequence]
print(str_1[1:5:2])   # start at 1 and then increment by 2 [start:stop:step]
```

```
el
Hel
lo
o
Hlo
el
```

4.2 Split and join strings

split() and **join()** methods allow creating a list of words from a string and formatted text from different words, respectively.

[15]:
```
str_1 = 'Hello, how are you?'
str_1.split() # this will split the string at spaces
```

[15]:
```
['Hello,', 'how', 'are', 'you?']
```

[16]:
```
str_1.split(',')   # this will split the string at a specific separator
```

[16]:
```
['Hello', ' how are you?']
```

We can reconstruct the above string using the join() method.

[18]:
```
','.join(str_1.split(','))
```

[18]:
```
'Hello, how are you?'
```

or use another separator instead of comma, let's say hyphen (-)

[19]:
```
'-'.join(str_1.split(','))
```

[19]:
```
'Hello- how are you?'
```

[20]:
```
multiline_str = '''This is a multi-line string
I\'ll display how to use the splitlines() method
to split all lines from the text and
store in a list'''
multiline_str.splitlines()
```

[20]:
```
['This is a multi-line string',
 "I'll display how to use the splitlines() method",
 'to split all lines from the text and ',
```

```
 'store in a list']
```

4.3 Find the lowest and highest index of a substring

We can use the **find()** or **rfind()** methods to find the lowest and highest index associated with a substring in a given string.

[23]:
```
str_1.find('o') # first time the character 'o' appears in str_1 is at index 4␣
↪(H->0, e->1, l->2, l->3, o->4)
```

[23]:
```
4
```

[22]:
```
str_1.rfind('o') # last time the character 'o' appears in str_1 ist at index 16
```

[22]:
```
16
```

4.4 Find and replace a substring

[24]:
```
str_2 = "Hello Jubran, are you playing soccer this summer?"
str_2.replace('soccer','rugby') # this will replace soccer with rugby in str_2
```

[24]:
```
'Hello Jubran, are you playing rugby this summer?'
```

4.5 How many times a substring appeared in a given string?

[25]:
```
str_2.count('o')
```

[25]:
```
3
```

4.6 Edit the case in a string

[27]:
```
str_3 = 'Python programming is fun!'
print(str_3.upper())  # all uppercase letters
print(str_3.lower())  # all lowercase letters
print(str_3.capitalize()) # only the first letter in the string will be uppercase
print(str_3.title())      # first letter of each word in the string will become␣
↪uppercase
print(str_3.swapcase())   # invert the case of each word in the original string
```

```
PYTHON PROGRAMMING IS FUN!
python programming is fun!
Python programming is fun!
```

```
Python Programming Is Fun!
pYTHON PROGRAMMING IS FUN!
```

4.7 Format a string

[28]:
```
str_4 = 'Section 1'
'{:-^50}'.format(str_4)   # horizontal line with string in the center
```

[28]:
```
'--------------------Section 1---------------------'
```

[29]:
```
'{:-<50}'.format(str_4)   # horizontal line with string on the left
```

[29]:
```
'Section 1----------------------------------------'
```

[30]:
```
'{:->50}'.format(str_4)   # horizontal line with string on the right
```

[30]:
```
'----------------------------------------Section 1'
```

[31]:
```
amount = 1000000    # display using thousand separtors
'{:,}'.format(amount)
```

[31]:
```
'1,000,000'
```

Summary

In this chapter, we learned about how to use strings and format texts in python.

Chapter 5

Operators

There exists a wide variety of operators. Here, we will only discuss the following operators:

1. Arithmetic operators
2. Comparison operators
3. Equality operators
4. Logical or Boolean operators

Other operators e.g. sequence, set and dictionary operators will be discussed in the later chapters.

5.1 Arithmetic operators

Common arithmetic operators include addition, subtraction, multiplication, division, modulo and exponentiation operators. Python allows two kind of division operators , one is called true division and the other is called integer or float division with two forward slashes.

[8]:
```
num_1, num_2 = 40, 15
print(f'Addition: {num_1} + {num_2} \t\t=\t {num_1 + num_2}')
print(f'Subtraction: {num_1} - {num_2} \t\t=\t {num_1 - num_2}')
print(f'Multiplication: {num_1} * {num_2} \t=\t {num_1 * num_2}')
print(f'True division: {num_1} / {num_2} \t\t=\t {num_1 / num_2:.2f}')
print(f'float division: {num_1} // {num_2} \t=\t {num_1 // num_2}')
print(f'Modulo: {num_1} % {num_2} \t\t=\t {num_1 % num_2}')
print(f'Exponentiation: {num_1} ** 2 \t=\t {num_1 ** 2}')
```

```
Addition: 40 + 15               =       55
Subtraction: 40 - 15            =       25
Multiplication: 40 * 15         =       600
True division: 40 / 15          =       2.67
float division: 40 // 15        =       2
Modulo: 40 % 15                 =       10
Exponentiation: 40 ** 2         =       1600
```

5.2 Comparison operators

The comparison operators allow us to see if one variable is greater or smaller than another variable.

[11]:
```
print(f'{num_1} is greater than {num_2}: {num_1 > num_2}')
print(f'{num_1} is greater than or equal to {num_2}: {num_1 >= num_2}')
print(f'{num_1} is smaller than {num_2}: {num_1 < num_2}')
print(f'{num_1} is smaller than or equal to {num_2}: {num_1 <= num_2}')
```

```
40 is greater than 15: True
40 is greater than or equal to 15: True
40 is smaller than 15: False
40 is smaller than or equal to 15: False
```

5.3 Equality operators

The equality operators allow us to see if two variables have similar identifiers or values.

[12]:
```
print(f'{num_1} has the same value as {num_2}: {num_1 == num_2}')
print(f'{num_1} has a different value than {num_2}: {num_1 != num_2}')
print(f'{num_1} has the same identifier as {num_2}: {num_1 is num_2}')
print(f'{num_1} has a different identifier {num_2}: {num_1 is not num_2}')
print(id(num_1), id(num_2))   # identities of num_1 and num_2
```

```
40 has the same value as 15: False
40 has a different value than 15: True
40 has the same identifier as 15: False
40 has a different identifier 15: True
140705061381744 140705061380944
```

5.4 Logical/Boolean operators

The logical or Boolean operators are used to see if either one or both conditions are true. The **not** operator returns the opposite value, for example, if a = True, **not a** will return the opposite of a which is false.

[13]:
```
print(num_1 > num_2 and num_1 > 20)        # both statements must be true for a␣
 ↪true output
print(num_1 < num_2 or num_1 > 20)         # atleast one statement should be true␣
 ↪for a true output
print(not num_1 > 20)                      # since num_1 > 20 is true as num_1 =␣
 ↪40, not num_1 > 20 must output false
```

```
True
True
False
```

5.5 Operator precedence (PEMDAS rule)

Python follows PEMDAS rule. PEMDAS is an acronym for paranthesis, exponents, multiplication, division, addition and subtraction. As per this rule, anything within paranthesis is evaluated first, then the exponents, and then multiplication and divisions (both have the same level, it does not matter if multiplication is done first or division), followed by addition and subtraction (both have the same level).

[14]:
```
exp_1 = (2 + 3)**2          # in this anything inside the parenthesis is␣
 ↪evaluated first, which equals 5 and then the exponent
exp_2 = 2 + 3**2            # in this exponent is evaluated first and then the␣
 ↪addition
print (exp_1, exp_2)        # exp_1 = 25, whereas exp_2 = 11, so it's important␣
 ↪to follow the correct operator precedence
```

```
25 11
```

Summary

In this chapter, we learned about arithmetic operators, comparison operators, equality operators, and boolean operators. These will be used a lot in routine computations and conditional programming. In addition, we learned about the PEMDAS rule, which python follows for operator precedence.

Chapter 6

Lists and tuples

A list is an order sequence of objects. It is a mutable data structure.

A list is created by placing all the items(elements) inside the **brackets[]**. These items are not required to be from the same data type, meaning we can store strings, integers, floats, another list, tuples, dictionary or even a None object within a list. In addition, lists are mutable data structures, which means we can change a list at any time by adding, removing, or changing its items.

A tuple is an order sequence of objects. It is an immutable data structure.

A tuple is created by place all the items(elements) inside the **parenthesis()**. Like lists, these items are not required to be from the same data type. Tuples are immutable data structures, which means that once we assign objects to a tuple, it cannot be changed under any circumstance.

Example 1 Change an item from the list of fruits –> [apples, bananas, oranges, grapes]

[2]:
```
list_1 = ['apples', 'bananas', 'oranges', 'grapes']       # this is a list of 4␣
 ↪items, index starts at 0, so [0, 1, 2, 3]
list_1[1] = 'avacados'   # changing bananas to avacados
print(list_1)
```

```
['apples', 'avacados', 'oranges', 'grapes']
```

see no problem in changing the items from a list since it's a mutable data structure. Now, let's try to do the same to a tuple.

[3]:
```
tuple_1 = ('apples', 'bananas', 'oranges', 'grapes')
tuple_1[1] = 'avacados'
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-3-34aad2d0de11> in <module>
  1 tuple_1 = ('apples', 'bananas', 'oranges', 'grapes')
----> 2 tuple_1[1] = 'avacados'
```

```
TypeError: 'tuple' object does not support item assignment
```

ooooops! cannot change items in tuples as they are immutable data structures.

Now you must be thinking that why do we use tuples then? Tuples are good for storing parameters that we do not want to change in your program. Tuples use less memory as compared to the lists and creating a tuple is faster than creating a list.

6.1 Create lists

[5]:
```
list_1 = list()          # create empty lists using list() constructor
list_2 = []              # create empty lists using brackets
list_3 = ['apples', 1.5, 2, [1, 3, 5], {'a':1, 'b':4}]   # lists can contain␣
 ↪different data types
print(list_1, list_2, list_3, sep='\n')
```

```
[]
[]
['apples', 1.5, 2, [1, 3, 5], {'a': 1, 'b': 4}]
```

6.2 Add items to a list

[15]:
```
list_1 = ['apples', 'oranges']
list_1.append('pears')                      # add pears at the end of the list
list_1.insert(1, 'avacados')                # use insert() method to add item at␣
 ↪a specific index position
print(list_1)
```

```
['apples', 'avacados', 'oranges', 'pears']
```

Let's say we want to add another list ([strawberries, blueberries, raspberries]) at the end of the list_1, we can do so by using the **extend()** method

[16]:
```
list_1.extend(['strawberries', 'blueberries', 'raspberries'])
print(list_1)
```

```
['apples', 'avacados', 'oranges', 'pears', 'strawberries', 'blueberries',
'raspberries']
```

6.3 Remove items from a list

We can use the **pop()** method to extract and remove an element at a specific index from a given list. If we do not specify the index value, **pop()** will remove the last element from the list.

[17]:
```
list_1.pop()                          # use the pop() method to remove the last item␣
 ↪in the list
```

[17]:
```
'raspberries'
```

[18]:
```
list_1.pop(1)                    # This will extract and remove avacados
print(list_1)
```

```
['apples', 'oranges', 'pears', 'strawberries', 'blueberries']
```

We can use the **remove()** method to remove the first occurrence of a given object in the list

[19]:
```
list_1.remove('oranges')
print(list_1)
```

```
['apples', 'pears', 'strawberries', 'blueberries']
```

We can also use the **del** key to remove an item at a given index from the list. In this method, we can even specify a range of indices to remove the corresponding items from the list.

[20]:
```
del list_1[1:3]
print(list_1)
```

```
['apples', 'blueberries']
```

Finally, we can use the **clear()** method to remove all items from the list. This will result in an empty list.

[21]:
```
list_1.clear()
print(list_1)
```

```
[]
```

6.4 Sorting lists

We can use the **sort()** method to sort the items of a list in ascending or descending order.

[23]:
```
list_1 = [1, 4, 7, 2, 9, 6]
list_1.sort()                # the sort() method if used without any parameters␣
 ↪results in an ascending sort
print(list_1)
```

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

[24]:
```
list_1 = [1, 4, 7, 2, 9, 6]
list_1.sort(reverse=True)    # descending sort
print(list_1)
```

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

[25]:
```
list_1.reverse()        # flips the order of list items
print(list_1)
```

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

6.5 Sequence operators

[26]:
```
list_1 = [1, 3, 5]
list_2 = [2, 4]
print(list_1 + list_2)     # use concatenation operator (+) to merge lists
```

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

[27]:
```
print(4*[2, 4])           # use multiplicity (*) to generate additional copies␣
 ↪of the list items
```

```
[2, 4, 2, 4, 2, 4, 2, 4]
```

[28]:
```
print(5 in list_1)        # use in or not in to check if an item exists in a list
print(5 not in list_2)
```

```
True
True
```

We can also use the slicing feature

[29]:
```
print(list_1[0], list_1[-1])    # first and last items of list_1
```

```
1 5
```

[30]:
```
print(list_1[:2], list_1[-2:])  # first two and last two items of list_1
```

```
[1, 3] [3, 5]
```

6.6 Create tuples

[31]:
```
tuple_1 = tuple()         # tuple constructor
tuple_2 = ()              # empty paranthesis
tuple_3 = ('apples', 1, 2, [1, 2, 3], {'a':1, 'b':4})
print(tuple_1, tuple_2, tuple_3, sep='\n')
```

```
()
()
('apples', 1, 2, [1, 2, 3], {'a': 1, 'b': 4})
```

We can convert a tuple to a list using the **list()** constructor, and a list to a tuple using the **tuple()** constructor.

[32]:
```
list(tuple_3)
```

[32]:
```
['apples', 1, 2, [1, 2, 3], {'a': 1, 'b': 4}]
```

6.7 Unpack a sequence

[33]:
```
a, b, c, d, e = tuple_3            # tuple_3 has 5 items
print(e)
```

```
{'a': 1, 'b': 4}
```

now what if we only want to get the first item as one variable and the remaining items as another variable

[34]:
```
a, b = tuple_3
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-34-abd212451719> in <module>
----> 1 a, b = tuple_3


ValueError: too many values to unpack (expected 2)
```

oooooooops! a ValueError: too many values to unpack —- we can correctly do this using an asterisk (*) before b

[35]:
```
a, *b = tuple_3
print(b)
```

```
[1, 2, [1, 2, 3], {'a': 1, 'b': 4}]
```

Let's say we are not interested in storing b, we can use a dummy/temporary variable (_) instead

[36]:
```
a, *_ = tuple_3
print(a)
```

```
apples
```

What if we are interested in getting the first two items and the last items only?

[37]:
```
a, b, *_, c = tuple_3
print(a, b, c, sep = '\n')
```

```
apples
1
{'a': 1, 'b': 4}
```

Summary

In this chapter, we learned how to create lists and tuples, and add, or remove items in the list or concatenate different lists. In addition, we learned about the sequence operators and unpacking sequences.

Chapter 7

Dictionaries

> **A dictionary** is a collection of key/value pairs, with the requirement that the keys are unique.

A dictionary is created by placing all the key:value pairs inside the **braces{}**.

Dictionaries are often called maps, hashmaps, lookup tables or associative arrays in other programming languages. These are mutable data structures, meaning we can easily add or remove items and update their values. Since the keys of a dictionary are required to be unique, adding a key–value item whose key is the same as an existing key will simply update the key's current value with the new value.

From Python 3.7, dictionaries preserve the insertion order by default. Updating the keys does not affect their order and if we delete a key and add it again, it will be inserted at the end.

We use keys to access their associated data values in the dictionaries. In the case of lists, we used numeric indices to access data, however dictionaries have no notion of index position and therefore, cannot be sliced.

Why are dictionaries useful? Instead of searching for a value in the entire list, we can extract any value from a dictionary using its associated key, thus we can efficiently search, insert, and delete any object associated with a given key.

7.1 Create dictionaries

[3]:
```
# empty dictionaries using the dict() constructor and empty braces
dict_1 = dict()
dict_2 = {}
# non-empty dictionaries
dict_3 = {'a': 1, 'b': 4}
# dictionaries from two lists (one contains keys, and other contains the␣
 ↪corresponding values)
# we can use the zip() function, which returns an iterator of tuples of␣
 ↪aggregrated items
key_1 = ['a', 'b', 'c', 'd']
```

```
value_1 = [2, 'apple', [1, 2, 3], (2, 5)]
dict_4 = dict(zip(key_1, value_1))
# using fromkeys() method, which requires keys as input, and a default value to␣
 ↪initialize each key, if not given, None
dict_5 = dict().fromkeys(key_1)   # initialize each key with None
dict_6 = dict().fromkeys(key_1, 0) # initialize each key with zeros

print(dict_1, dict_2, dict_3, dict_4, dict_5, dict_6, sep='\n')
```

```
{}
{}
{'a': 1, 'b': 4}
{'a': 2, 'b': 'apple', 'c': [1, 2, 3], 'd': (2, 5)}
{'a': None, 'b': None, 'c': None, 'd': None}
{'a': 0, 'b': 0, 'c': 0, 'd': 0}
```

7.2 Access values

[4]:
```
dict_1 = {'a': 2, 'b': 'apple', 'c': [1, 2, 3], 'd': (2, 5)}
dict_1['d']
```

[4]:
```
(2, 5)
```

[5]:
```
dict_1['e']
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-5-911708c4fe99> in <module>
----> 1 dict_1['e']


KeyError: 'e'
```

ooooops! KeyError — we can avoid this using the **get()** method. If the key is not available, it will return None. We can also customize the default value.

[7]:
```
print(dict_1.get('e'))
```

```
None
```

[8]:
```
print(dict_1.get('e', 'not available'))
```

```
not available
```

7.3 Add/remove and update

[10]:
```
dict_1 = {'a': 2, 'b': 'apple', 'c': [1, 2, 3], 'd': (2, 5)}
# adding new values
dict_1['e'] = 3.2
print(dict_1)
```

```
{'a': 2, 'b': 'apple', 'c': [1, 2, 3], 'd': (2, 5), 'e': 3.2}
```

We can also use the **update()** method to update values of existing keys and add new key:value pairs

[11]:
```
dict_1.update({'a': 4, 'f': 'soccer'})
print(dict_1)
```

```
{'a': 4, 'b': 'apple', 'c': [1, 2, 3], 'd': (2, 5), 'e': 3.2, 'f': 'soccer'}
```

To remove a key, we can use the **pop()** method

[12]:
```
dict_1.pop('f')
print(dict_1)
```

```
{'a': 4, 'b': 'apple', 'c': [1, 2, 3], 'd': (2, 5), 'e': 3.2}
```

[13]:
```
dict_1.pop('f')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)

<ipython-input-13-99356867615f> in <module>
----> 1 dict_1.pop('f')


KeyError: 'f'
```

ooooops! a KeyError . . . since we have already removed this key. To avoid this error, we can set a default message to display

[14]:
```
dict_1.pop('f', 'does not exist')
```

[14]:
```
'does not exist'
```

To remove the last key:value pair from the dictionary, we can use the **popitem()** method

[15]:
```
dict_1.popitem()
print(dict_1)
```

```
{'a': 4, 'b': 'apple', 'c': [1, 2, 3], 'd': (2, 5)}
```

We can also use the **del** key to delete a key from the dictionary

[16]:
```
del dict_1['c']
print(dict_1)
```

```
{'a': 4, 'b': 'apple', 'd': (2, 5)}
```

To remove all key:value pairs from the dictionary, we can use the **clear()** method. This will result in an empty dictionary.

[17]:
```
dict_1.clear()
print(dict_1)
```

```
{}
```

7.4 Dictionary operators

[18]:
```
dict_1 = {'a': 1, 'b': 4}
dict_2 = {'a': 1, 'b': 4}
print(dict_1 == dict_2)
```

```
True
```

[19]:
```
# now we update key 'a' in dict_2
dict_2.update({'a': 4})
print(dict_1 == dict_2)
```

```
False
```

[20]:
```
print(dict_1 != dict_2)          # not equal operator
```

```
True
```

[21]:
```
print('a' in dict_1)           # check if a key exists in a dictionary
```

```
True
```

[22]:
```
dict_1.keys()           # show all keys in the dictionary
```

[22]:
```
dict_keys(['a', 'b'])
```

[23]:
```
dict_1.values()        # show values in the dictionary
```

[23]:
```
dict_values([1, 4])
```

[24]:
```
dict_1.items()        # show key, value tuples in the dictionaries
```

[24]:
```
dict_items([('a', 1), ('b', 4)])
```

[25]:
```
# to check the number of key:value pairs in the dictionary we can use the len()␣
 ↪method
len(dict_1)
```

[25]:
```
2
```

Summary

In this chapter, we learned how to create dictionaries, access values, add/remove or update key-values. In addition, we learned about different dictionary operators.

Chapter 8

Sets

A set is an unordered collection of unique hashable objects as its elements.

A set is created by placing all the elements inside the **braces{}**.

Hashable objects have a corresponding hash value which never changes during their lifetime. Python has a builtin **hash()** function to find the hash value of an object, if it's hashable. Most of Python's immutable built-in objects (such as int, floats, strings) are hashable whereas mutable containers (such as lists or dictionaries) are not hashable.

A set, itself, is a mutable data structure, meaning we can easily add or remove items. Since sets are unordered collections, they have no notion of index position and therefore, cannot be sliced.

Python also offers an immutable set type, called frozenset.

Why are sets useful? Sets can be used to efficiently remove duplicate values from a list or tuple (as sets can only store unique elements) and also to perform common math operations like unions and intersections (Figure 8.1).

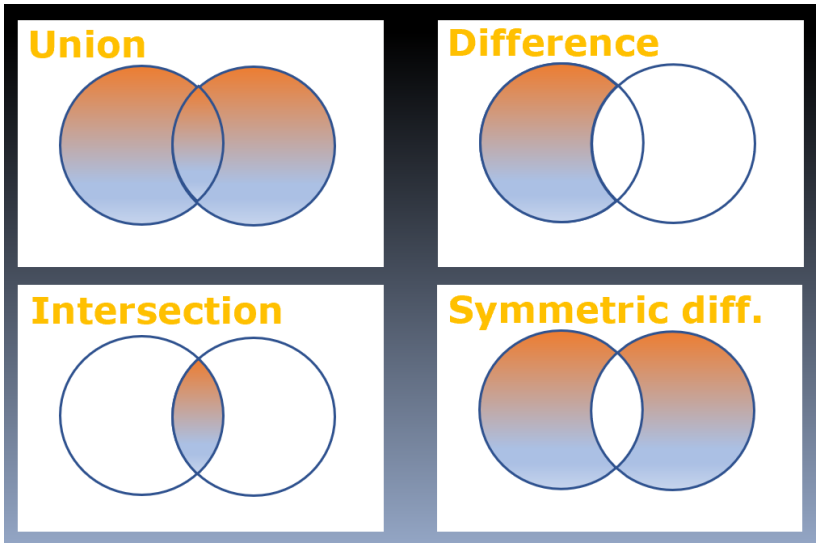


Figure 8.1: Examples of common set operations.

8.1 Create sets

[2]:
```
# The only way to create an empty set is using the set() constructor. We cannot␣
↪use empty {} as this will create a dictionary.
set_1 = set()
print(set_1)
```

```
set()
```

[3]:
```
a = {}
type(a)
```

[3]:
```
dict
```

[4]:
```
# we can create non-empty sets using braces
set_2 = {1, 3, 5, 7, 5}
print(set_2)
```

```
{1, 3, 5, 7}
```

what just happened .... you can include repeated values, however set will only store unique elements.

We can also use the set() constructor with iterables

[5]:
```
set_3 = set('aeiou')
print(set_3)
```

```
{'e', 'o', 'i', 'u', 'a'}
```

[6]:
```
set_4 = set([2, 4, 6, 8])
print(set_4)
```

```
{8, 2, 4, 6}
```

don't worry .. set is an unordered collection

8.2 Add elements or another set

[7]:
```
print(set_1)
```

```
set()
```

[8]:
```
set_1.add(2)
print(set_1)
```

```
{2}
```

[9]:
```
set_1.add(4)
print(set_1)
```

```
{2, 4}
```

To add another set, we use the **update()** method

[10]:
```
set_1.update(set_2)
print(set_1)
```

```
{1, 2, 3, 4, 5, 7}
```

8.3 Remove elements

[11]:
```
set_1.remove(3)      # specify the element to be removed
print(set_1)
```

```
{1, 2, 4, 5, 7}
```

[12]:
```
# we can also use the discard() method
set_1.discard(2)   # specify the element to be removed
print(set_1)
```

```
{1, 4, 5, 7}
```

[13]:
```
# or we can use the pop() method
set_1.pop()
print(set_1)
```

```
{4, 5, 7}
```

8.4 Set operations

[14]:
```
# union
print(set_1.union(set_2))
# another way
print(set_1 | set_2)
```

```
{1, 3, 4, 5, 7}
{1, 3, 4, 5, 7}
```

[15]:
```
# intersection
print(set_1.intersection(set_2))
# another way
print(set_1 & set_2)
```

```
{5, 7}
{5, 7}
```

[16]:
```
# difference
print(set_1.difference(set_2))
# another way
print(set_1 - set_2)
```

```
{4}
{4}
```

[17]:
```
# check membership
print(set_1)
print(4 in set_1)
print(4 not in set_1)
```

```
{4, 5, 7}
True
False
```

[19]:
```
# check if a set is subset of another set
set_2 = {3, 4, 5, 6, 7, 8}
print(set_1.issubset(set_2))
# another way
print(set_1 <= set_2)
```

```
True
True
```

Summary

In this chapter, we learned about creating empty and non-empty sets, adding/removing elements and updating sets. In addition, we learned about different set operations.

Chapter 9

if-elif-else

Often times, we make decisions based on the situations around us.

Example: We stop by a convenience store and want to buy a bag of chips. However, we don't want any bag of chips but a specific flavor made by a specific brand. We have purchased these chips from the main grocery store before and we know that it costs a certain amount. Since we do not want to overspend, we will only buy these chips if the price is within a certain range.

This is a decision-making scenario – we'll buy a bag of chips only if all of the requirements are met.

Figure 9.1 demonstrates how to think like a programmer. We go to a convenience store and I assume that we found the correct aisle containing different varieties of chips.

1. We'll first check if the brand of choice exists. If the convenience store does not have the specific brand of chips, a decision is made "do not buy" – but if the specific brand of chips exist, we'll check the next condition.
2. If the specific brand of chips exist, we'll check the next requirement/condition i.e. do we see the specific flavor? Again, if the condition is not met, we will make our decision "do not buy".
3. If we found both the right brand and flavor of chips, we will check the third requirement (price). Now if the price range is what we were hoping, we will buy the bag of chips. Otherwise, no chips :(

In python programming, we write these decision-control instruction using if-elf-else statements.

"if statement" is written as:

```
if condition:
    statement(s)
```

"if-else statement" is written as:

```
if condition:
    statement(s)
else:
    statement(s)
```

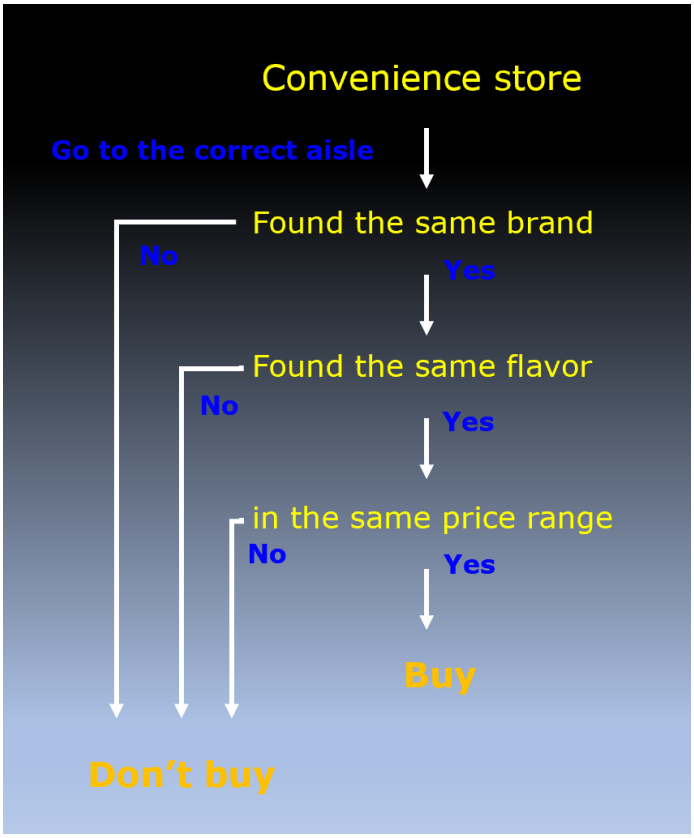


Figure 9.1: A flowchart to explain the decision-control instructions for the above example (buy a bag of chips from a convenience store).

"if-elif-else statement" is written as:

```
if condition 1:
    statement(s)
elif condition 2:
    statement(s)
else:
    statement(s)
```

Let's look more examples in which we determine pass/fail or letter grade from the percentage score.

Example 1 Given the percentage score, find out whether a person passed.

Passing criterion: percentage score must be greater than or equal to 40

[8]:
```
percentage_score = 75

if percentage_score < 40:
    print("Sorry! Try again.")
else:
    print("Congratulations! You passed.")
```

```
Congratulations! You passed.
```

Ternary operators allow us to write the above multi-line if-else statement in one line > statement if **condition** else statement

[9]:
```
print("Congratulations! You passed.") if percentage_score >= 40 else␣
 ↪print("Sorry, Try again.")
```

```
Congratulations! You passed.
```

Example 2 Given the percentage score, find the corresponding letter grade.

Conversion chart:

Percentage score	Letter grade
90-100	A+
80-89	A
75-79	B+
70-74	B
65-69	B-
60-64	C+
55-59	C
50-54	C-
40-49	D
< 40	F

[5]:
```
percentage_score = 65
if 90 <= percentage_score <= 100:
    print("The letter grade is A+")
elif 80 <= percentage_score <= 89:
    print("The letter grade is A")
elif 75 <= percentage_score <= 79:
    print("The letter grade is B+")
elif 70 <= percentage_score <= 74:
    print("The letter grade is B")
elif 65 <= percentage_score <= 69:
    print("The letter grade is B-")
elif 60 <= percentage_score <= 64:
    print("The letter grade is C+")
elif 55 <= percentage_score <= 59:
    print("The letter grade is C")
elif 50 <= percentage_score <= 54:
    print("The letter grade is C-")
elif 40 <= percentage_score <= 49:
    print("The letter grade is D")
else:
    print("The letter grade is F")
```

```
The letter grade is B-
```

Alternative Solution – instead of using the full interval notation, we will use ">=" in the conditions

[12]:
```
percentage_score = 95
if percentage_score >= 90:
    print("The letter grade is A+")
elif percentage_score >= 80:
    print("The letter grade is A")
elif percentage_score >= 75:
    print("The letter grade is B+")
elif percentage_score >= 70:
    print("The letter grade is B")
elif percentage_score >= 65:
    print("The letter grade is B-")
elif percentage_score >= 60:
    print("The letter grade is C+")
elif percentage_score >= 55:
    print("The letter grade is C")
elif percentage_score >= 50:
    print("The letter grade is C-")
elif percentage_score >= 40:
    print("The letter grade is D")
else:
    print("The letter grade is F")
```

```
The letter grade is A+
```

Summary

In this chapter, we learned about writing decision-control instructions in python using if-elif-else statements.

Chapter 10

Loops

There are two main programming structures: 1. Conditional Programming 2. Looping

Here we will discuss the looping structure in detail. >**Looping** allows us to iterate over the same block of code many times. > Python offers two distinct looping constructs: 1. while loops 2. for loops

10.1 While loops

A **while loop** is based upon the repeated testing of a Boolean condition. It will keep running and executing the same block of code as long as the condition is **True** or a **break** keyword is met which allows to bring the program control out of the loop.

In python programming, a "while loop" is written as follows:

```
while condition:
    statement(s)
```

Let's look at some examples to see how while loops work.

Example 1 Print **Hello World** ten times.

[3]:
```
max_iter = 10                    # number of times we want to print "Hello World!"
counter = 1                      # counter is set to 1, because I want to print␣
 ↪the numbers [1, 2, 3, 4, .., 10]
while counter <= max_iter:       # the loop will print "Hello World!" as long as␣
 ↪counter is smaller than or equal to max_iter
    print(counter, "Hello World!", sep = '\t')
    counter += 1                 # increase the counter by 1. it is very␣
 ↪important --if not written ---> infinite loop
```

```
1       Hello World!
2       Hello World!
3       Hello World!
4       Hello World!
```

```
5       Hello World!
6       Hello World!
7       Hello World!
8       Hello World!
9       Hello World!
10      Hello World!
```

Example 2 Find out if an object(value) exists in a sequence.

[6]:
```
random_num = [2, 7, 4, 5, 9, 3, 6]              # list of random numbers
target_num = 2                                  # target number that we want to␣
 ↪find in the sequence
counter = 0
while counter < len(random_num):                # condition: run the loop as long␣
 ↪as counter is smaller than length of sequence
    if random_num[counter] == target_num:       # check if the random_num at index␣
 ↪position=counter is equal to target_num
        print("The object exists ...")
        break                                   # if the current random_num ==␣
 ↪target_num, stop and come out of the while loop
    counter += 1
```

```
The object exists ...
```

If the target number is not in the sequence, the above code does not print any statement to inform the user

The next example use the **else** keyword to print the statement "the object does not exist . . . "

Example 3 Find out if an object(value) exists or does not exist in a sequence.

[7]:
```
random_num = [2, 7, 4, 5, 9, 3, 6]
target_num = 10
counter = 0
while counter < len(random_num):
    if random_num[counter] == target_num:
        print("The object exists...")
        break
    counter += 1
else:                          # else will only execute if the while loop ends␣
 ↪normally i.e. not with the break statement.
    print("The object does not exist...")
```

```
The object does not exist...
```

Now, what if we are not interested in using counters and else in while loops . . . can we write the above program using while loops?

Yes,. . . .. , we can. . . an alternative solution is given as follows:

[10]:
```
random_num = [2, 7, 4, 5, 9, 3, 6]
iter_num = iter(random_num)          # get an iterator
target_num = 10

while True:                          # infinite loop, will stop with a break␣
↪statement
    num = next(iter_num, None)       # next function only takes one element from␣
↪the sequence, each time it is called.
    if num is None:
        print("The object does not exist...")
        break
    elif num == target_num:
        print("The object exists...")
        break
```

```
The object does not exist...
```

10.2 For loops

A **for loop** is useful for iteration of values from an iterable (e.g. string, tuple, list, . . .).

In python programming, a "for loop" is written as follows:

```
for var in iterator:
    statement(s)
```

Let's look at some examples of how for loops are used in python . . .

Example 1 Find out if an object(value) exists or does not exist in a sequence.

[13]:
```
random_num = [2, 7, 4, 5, 9, 3, 6]
target_num = 10

for num in random_num:  # each iteration, num picks a value in random_num -- 1st␣
↪value at 1st iter -- 2nd value at 2nd iter, ...
    if num == target_num:        # (continuation of the above comment)  till the␣
↪last value in the iterable.
        print("The object exists...")
        break
else:
    print("The object does not exist...")
```

```
The object does not exist...
```

Now let's say that we are interested in find the corresponding index as well, if the value exists. Let's look at the next example, in which we use the enumerate function to print the index value corresponding to a given value. > **enumerate**(iterable, start = 0) is a built-in function in python.

Example 2 Find out if an object(value) exists in a sequence. If yes, find the corresponding index.

[15]:
```
random_num = [2, 7, 4, 5, 9, 3, 6]
target_num = 3

for index, num in enumerate(random_num): #enumerate() can loop over an iterable␣
 ↪and produce an automatic counter.
    if num == target_num:
        print(f"The object exists at index {index}.")
        break
else:
    print("The object does not exist...")
```

```
The object exists at index 5.
```

Now let's say we are interested in printing the sequence in reverse order along with the corresponding indices ...

textbfExample 3 Print the index and values of a sequence a. in reverse order b. offset the index value by 3, meaning start = 3 in enumerate()

[16]:
```
random_num = [2, 7, 4, 5, 9, 3, 6]
sample_length = len(random_num)
# 3a ..........................................................................
# enumerate(iterable, starting_index), default starting index is 0
# we can use the reversed() function to flip the sequence
for index, num in enumerate(reversed(random_num)):
    print(sample_length-index, num, sep='-->')

print(50*"--")
# 3b ..........................................................................
for index, num in enumerate(random_num, 3):
    print(index, num, sep='-->')
```

```
7-->6
6-->3
5-->9
4-->5
3-->4
2-->7
1-->2
----------------------------------------------------------------------------------
--------------------
3-->2
4-->7
5-->4
6-->5
7-->9
8-->3
9-->6
```

Example 4 Print the element-by-element sum of three lists.

[17]:
```
list_1 = [2, 7, 4, 5, 9, 3, 6]
list_2 = [1, 2, 4, 3, 7, 9, 6]
list_3 = [9, 6, 8, 5, 4, 3, 7]
sample_length = len(list_1)
# element_by-element sum_of_lists ..........................................
# use zip(iterable_1, iterable_2, iterable_3, ...) to make an iterator that␣
 ↪aggregates elements from each of the iterables.
list_sum = []
for vals in zip(list_1, list_2, list_3):
    sum_vals = sum(vals)
    list_sum.append(sum_vals)
    print(f"{vals[0]} + {vals[1]} + {vals[2]} ------> {sum_vals}")
```

```
2 + 1 + 9 ------> 12
7 + 2 + 6 ------> 15
4 + 4 + 8 ------> 16
5 + 3 + 5 ------> 13
9 + 7 + 4 ------> 20
3 + 9 + 3 ------> 15
6 + 6 + 7 ------> 19
```

Example 5 Print the keys and their correspondig values from a dictionary

[18]:
```
# the phone number is just an aribtrary number
dict_1 = {'name': 'Michael',
          'age': 32,
          'sports': 'rugby',
          'phone': '000-000-5555'}

# dict.items() to get key/value pairs
for key, val in dict_1.items():
    print(key, "\t\t------>\t\t", val)
```

```
name            ------>         Michael
age             ------>         32
sports          ------>         rugby
phone           ------>         000-000-5555
```

often, we are interested in accessing a value in the sequence by its index Let's see another example in which we use range(), which is an immutable sequence type in python.

Example 6 Use the range() function to print the index and corresponding values of a list.

[20]:
```
list_1 = [2, 7, 4, 5, 9, 3, 6]

# range(start, stop, step), default step = 1
for count in range(len(list_1)):
```

```
    print(count, "\t\t------>\t\t", list_1[count])

print(50*"--")
# starting from index = 4
for count in range(4, len(list_1)):
    print(count, "\t\t------>\t\t", list_1[count])

print(50*"--")
# print everything with an interval of 2
for count in range(0, len(list_1), 2):
    print(count, "\t\t------>\t\t", list_1[count])
```

```
0               ------>         2
1               ------>         7
2               ------>         4
3               ------>         5
4               ------>         9
5               ------>         3
6               ------>         6
----------------------------------------------------------------------------------
--------------------
4               ------>         9
5               ------>         3
6               ------>         6
----------------------------------------------------------------------------------
--------------------
0               ------>         2
2               ------>         4
4               ------>         9
6               ------>         6
```

Summary

In this chapter, we learned about the while and for loops using many examples. We also used len(), iter(), next(), enumerate() and zip() functions. In addition, we have used the print() to show different ways of presenting the text.

Chapter 11

Comprehensions

A comprehension is a simple construct, that is sometimes referred to as syntactic sugar.

> In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. (Wikipedia)

These comprehensions are shortcuts designed for frequently used functionality to improve the code readibility and make it more concise.

Let's look at some examples . . .

Example 1 Use list comprehensions to find the square of each element in a sequence.

[4]:
```
list_1 = [2, 7, 4, 5, 9, 3, 6]
# plain for loop and list.append()-----------------------------------------------
squares = []
for val in list_1:
    square_val = val*val
    squares.append(square_val)
    print('square of '+ str(val), "\t\t------>\t\t", square_val)

print(50*"--")
# using list comprehensions------------------------------------------------------
squares_1 = [val*val for val in list_1]
[print(square, '\t=\t', square_1) for square, square_1 in zip(squares,␣
 ↪squares_1)]
```

```
square of 2             ------>         4
square of 7             ------>         49
square of 4             ------>         16
square of 5             ------>         25
square of 9             ------>         81
square of 3             ------>         9
square of 6             ------>         36
----------------------------------------------------------------------------------
--------------------
```

```
4       =       4
49      =       49
16      =       16
25      =       25
81      =       81
9       =       9
36      =       36
```

[4]:
```
[None, None, None, None, None, None, None]
```

Example 2 Use list comprehensions to update a dictionary.

[9]:
```
# update values of some keys in the dictionary
list_1 = [2, 7, 4, 5]
list_2 = ['a', 'b', 'c', 'd']

update_key = ['b', 'd']
update_val = [23, 12]
# plain for loop ------------------------------------------------------------
dict_1 = dict(zip(list_2, list_1))
print("Case: plain for loop -- just created a dictionary", dict_1, sep='\n')
for key, val in zip(update_key, update_val):
    dict_1[key] = val
print("Case: plain for loop -- updated the dictionary", dict_1, sep='\n')

print(50*"--")
# using the list comprehensions---------------------------------------------
dict_1 = dict(zip(list_2, list_1))
print("Case: comprehensions -- just created a dictionary", dict_1, sep='\n')
[dict_1.update({key: val}) for key, val in zip(update_key, update_val)]
print("Case: comprehensions -- updated the dictionary", dict_1, sep='\n')
```

```
Case: plain for loop -- just created a dictionary
{'a': 2, 'b': 7, 'c': 4, 'd': 5}
Case: plain for loop -- updated the dictionary
{'a': 2, 'b': 23, 'c': 4, 'd': 12}
----------------------------------------------------------------------------
--------------------
Case: comprehensions -- just created a dictionary
{'a': 2, 'b': 7, 'c': 4, 'd': 5}
Case: comprehensions -- updated the dictionary
{'a': 2, 'b': 23, 'c': 4, 'd': 12}
```

Example 3 Use dictionary comprehensions to create a new dictionary from another dictionary with same keys but squared values.

[8]:
```
list_1 = [2, 7, 4, 5]
list_2 = ['a', 'b', 'c', 'd']
```

```
dict_1 = dict(zip(list_2, list_1))
# create a new dictionary with a square of values in list_1
dict_2 = {key:val*val for key, val in dict_1.items()}
print("original dictionary 1", dict_1, sep='\n')
print("new dictionary containing square values using comprehensions",
      dict_2, sep='\n')
```

```
original dictionary 1
{'a': 2, 'b': 7, 'c': 4, 'd': 5}
new dictionary containing square values using comprehensions
{'a': 4, 'b': 49, 'c': 16, 'd': 25}
```

Summary

In this chapter, we learned about how to make our python code more concise using comprehension constructs.

Chapter 12

Functions

As programs get bigger and more complicated, they become more difficult to read and debug. To prevent redundant logic, manage code complexity and improve the code readability, the recommended practice is to write parts of code as functions that can be reused without writing the entire block of code.

> **A function** is a group of related statements that can be called together to perform a specific task.

In python, a function is typically written as:

```
def function_name(input=optional):
    statement(s)
    return output(optional)
```

As you can see, a function may or may not take input parameters, and also may or may not return an output value.

Example 1: Write a function to say Hello.

[1]:
```
def say_hello():          # this function does not take any input parameters
    print("I'm a function and I say Hello.")       # this function also does not␣
↪return any output value
```

[2]:
```
say_hello()             # calling say_hello() function
```

```
I'm a function and I say Hello.
```

Example 2: Write a function to say Hello to a person (input parameter).

[3]:
```
def hello(name):           # this function takes name as input parameter
    print(f'Hello {name}, how are you?')
```

[4]:
```
hello('Jubran')       # calling function
hello('Jordan')
```

```
Hello Jubran, how are you?
Hello Jordan, how are you?
```

Example 3: Write a function to convert temperature measurements from Fahrenheit to Celsius.

[5]:
```
def fahrenheit_to_celsius(temp_fahrenheit):
    return (temp_fahrenheit - 32)/1.8           # this function returns an␣
 ↪output value (temperature in Celsius)
```

[8]:
```
print(f'{fahrenheit_to_celsius(-32):.2f}')
```

```
-35.56
```

[9]:
```
celsius_temp = fahrenheit_to_celsius(-32)     # we can store the output from a␣
 ↪function
print(f'{celsius_temp:.2f}')
```

```
-35.56
```

Example 4: If a project has a earned value of $10,000 but actual costs were $9,500, find the cost performance index (CPI)

[10]:
```
def cost_performance_index(earned_value, actual_cost):      # this function has␣
 ↪two input parameters
    return earned_value/actual_cost
```

[12]:
```
cpi = cost_performance_index(10000, 9500)
if cpi >= 1:
    print(f"CPI = {cpi:.2f}: project is under/on budget -- no need to change␣
 ↪anything")
else:
    print(f"CPI = {cpi:.2f}: project is over budget -- do something!")
```

```
CPI = 1.05: project is under/on budget -- no need to change anything
```

Example 5: Find the area of a rectangle with length = 10m, width = 5m.

[13]:
```
def rectangle_area(length, width=3):       #setting default value for width
    return length*width
```

[14]:
```
print(rectangle_area(10, 5))
```

```
50
```

[15]:
```
print(rectangle_area(10))      #it will still work as the function will use the␣
 ↪default value for width, which is 3m
```

```
30
```

Example 6: Write a function with positional arguments.

[16]:
```
def sum_of_squares(*args):                #it allows varying number of positinal␣
 ↪arguments
    return sum([arg**2 for arg in args])
```

[17]:
```
print(sum_of_squares(1,3,5,7))      # answer should be 1+9+25+49 = 84
```

```
84
```

Example 7: Write a function with keyword arguments.

[19]:
```
def favourite_sports(**kwargs):
    print(f"{kwargs['name']} likes {kwargs['sports']}")
```

[21]:
```
dict_1 = {'name':'Jubran', 'sports': 'soccer'}
favourite_sports(**dict_1)
```

```
Jubran likes soccer
```

[22]:
```
dict_1.update({'name':'Tim', 'sports': 'rugby'})
favourite_sports(**dict_1)
```

```
Tim likes rugby
```

12.1 Anonymous functions

In python programming, **lambda** keyword is used to declare anonymous (inline) functions, for example

[23]:
```
polynomial = lambda x: x**2 + 2*x + 1
```

[24]:
```
print(polynomial(5))
```

```
36
```

Summary

In this chapter, we learned about writing functions that help in managing code complexity and improving code readablity.

Chapter 13

Import modules

For easier maintenance, it is important to split a longer program into several files that can be imported later into a main program. Like functions, modules also help in improving code readability and managing code complexity.

A module is a file containing Python definitions and statements.

It's easier to write modules in python. For example, I wrote the following code in a file and saved it as **temp_calc.py**. This is a module, which I can import for use, when required.

[1]:
```
def fahrenheit_to_celsius(temp_fahrenheit):
    return (temp_fahrenheit - 32)/1.8

def celsius_to_fahrenheit(temp_celsius):
    return 1.8*temp_celsius + 32
```

In addition to functions, we can also specify constants, variables, or classes in the modules.

13.1 Import modules

Let's say I want to use some function from the **temp_calc.py** module, I'll import it as follows:

Example 1 Convert a temperature measurement (97oF) from Fahrenheit to Celsius.

[5]:
```
import temp_calc          #explicit module import
print(f'{temp_calc.fahrenheit_to_celsius(97):.2f}\u00b0C')
```

```
36.11°C
```

we can also import a module explicitly by alias, for example

[6]:
```
import temp_calc as tc
print(f'{tc.fahrenheit_to_celsius(97):.2f}\u00b0C')
```

```
36.11°C
```

we can also import particular items from a module

[7]:
```
from temp_calc import fahrenheit_to_celsius          # this way, we can simply␣
 ↪call the function
print(f'{fahrenheit_to_celsius(97):.2f}\u00b0C')
```

```
36.11°C
```

Let's say we want to import all of the module contents into the local namespace.

[8]:
```
from temp_calc import *
print(f'{fahrenheit_to_celsius(97):.2f}\u00b0C')
```

```
36.11°C
```

When writing programs in Python, we often use/import many built-in modules such as **math**, **datetime** ... or third-party modules such as **numpy**, **matplotlib**, **pandas**, ... so you will be using these or similar commands a lot in your programs

[9]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

Summary

In this chapter, we learned about writing modules in Python, as well as importing modules in our programs for use, when required.

Chapter 14

Decorators

Decorators are design patterns that augment the behavior of an existing object, thus decorating the original behavior.

Let's look at the following function

[1]:
```
def compute_polynomial(x):
    return x**2 + 2*x + 1
```

we run this function for one element, it works

[2]:
```
print(compute_polynomial(5))       # 5**2 + 2*5 +1 = 25+10+1 = 36
```

```
36
```

We can also use the **map()** function to do element-by-element computation for an input list

[4]:
```
x = [1, 2, 3, 4, 5]
y = map(compute_polynomial, x)
print(list(y))
```

```
[4, 9, 16, 25, 36]
```

or use list comprehensions to evaluate the function for each element in the list

[5]:
```
print([compute_polynomial(val) for val in x])
```

```
[4, 9, 16, 25, 36]
```

or we can write a decorator, for which we need to define a function that takes another function as the input parameter and returns a new function... let's say, we write a list_mapper

[6]:
```
def list_mapper(fun):       # it takes a function input = fun
    def inner(*args):       # define an inner function
        if len(args)==1:
            return fun(*args)
        elif len(args) > 1:
```

```
            return [fun(arg) for arg in args]
        else:
            raise TypeError("Missing arguments")
    return inner            # return the inner function
```

now, we can decorate our compute_polynomial function as

[7]:
```
@list_mapper
def compute_polynomial(x):
    return x**2 + 2*x + 1
```

[9]:
```
print(compute_polynomial(*x))          # no need to write more code again and␣
 ↪again
```

```
[4, 9, 16, 25, 36]
```

we can use the same decorator/wrapper on other functions, such as

[10]:
```
@list_mapper
def squares(x):
    return x**2
```

[11]:
```
print(squares(*x))
```

```
[1, 4, 9, 16, 25]
```

Summary

In this chapter, we learned about decorators that augment the behavior of an existing object.

Chapter 15

Read/write files

Often times, we are required to read data from files (text, csv, json, ...) or save data to these files in our programs. Let's see how to read/write files in python:

Example 1 Write a text file, storing greetings to ten different persons and then read it.

[3]:
```
names = ['Jubran', 'Aamir', 'John', 'Michael', 'David', 'Maryam', 'Christina',␣
 ↪'Maria', 'Jordan', 'Jasmine']
# with context manager, we don't need to worry about closing the file.
with open('greetings.txt', 'w') as file:            # mode, w = write , a =␣
 ↪append, r= read
    [file.write(f'Hello {name}, how are you?\n') for name in names]
```

Now let's read this file ...

[5]:
```
with open('greetings.txt', 'r') as file:             # mode, w = write , a =␣
 ↪append, r= read
    print(file.read())
```

```
Hello Jubran, how are you?
Hello Aamir, how are you?
Hello John, how are you?
Hello Michael, how are you?
Hello David, how are you?
Hello Maryam, how are you?
Hello Christina, how are you?
Hello Maria, how are you?
Hello Jordan, how are you?
Hello Jasmine, how are you?
```

we can also read by character limit, for example ...

[6]:
```
with open('greetings.txt', 'r') as file:             # mode, w = write , a =␣
 ↪append, r= read
```

```
    print(file.read(100))                    # read first 100 characters only
```

```
Hello Jubran, how are you?
Hello Aamir, how are you?
Hello John, how are you?
Hello Michael, how are
```

To read all lines and store them as items in a list, we can use the readlines() method ...

[7]:
```
with open('greetings.txt', 'r') as file:              # mode, w = write , a =␣
↪append, r= read
    print(file.readlines())
```

```
['Hello Jubran, how are you?\n', 'Hello Aamir, how are you?\n', 'Hello John, how
are you?\n', 'Hello Michael, how are you?\n', 'Hello David, how are you?\n',
'Hello Maryam, how are you?\n', 'Hello Christina, how are you?\n', 'Hello Maria,
how are you?\n', 'Hello Jordan, how are you?\n', 'Hello Jasmine, how are
you?\n']
```

we can remove the newline character (\n) using splitlines() with read, if required ...

[11]:
```
with open('greetings.txt', 'r') as file:              # mode, w = write , a =␣
↪append, r= read
    print(file.read().splitlines())
```

```
['Hello Jubran, how are you?', 'Hello Aamir, how are you?', 'Hello John, how are
you?', 'Hello Michael, how are you?', 'Hello David, how are you?', 'Hello
Maryam, how are you?', 'Hello Christina, how are you?', 'Hello Maria, how are
you?', 'Hello Jordan, how are you?', 'Hello Jasmine, how are you?']
```

Example 2 Load a csv file (I have a csv file downloaded from https://people.sc.fsu.edu/~jburkardt/data/csv/biostats.csv).

[12]:
```
import csv
with open('biostats.csv', newline='') as csvfile:
     csv_file = csv.reader(csvfile, delimiter=',')
     for row in csv_file:
         print('\t|\t'.join(row))
```

```
Name    |           "Sex"       |        "Age"  |        "Height (in)"  |
"Weight (lbs)"
Alex    |             "M"       |          41   |              74       |
170
Bert    |             "M"       |          42   |              68       |
166
Carl    |             "M"       |          32   |              70       |
155
Dave    |             "M"       |          39   |              72       |
167
```

```
Elly    |             "F"      |          30   |             66       |
124
Fran    |             "F"      |          33   |             66       |
115
Gwen    |             "F"      |          26   |             64       |
121
Hank    |             "M"      |          30   |             71       |
158
Ivan    |             "M"      |          53   |             72       |
175
Jake    |             "M"      |          32   |             69       |
143
Kate    |             "F"      |          47   |             69       |
139
Luke    |             "M"      |          34   |             72       |
163
Myra    |             "F"      |          23   |             62       |
98
Neil    |             "M"      |          36   |             75       |
160
Omar    |             "M"      |          38   |             70       |
145
Page    |             "F"      |          31   |             67       |
135
Quin    |             "M"      |          29   |             71       |
176
Ruth    |             "F"      |          28   |             65       |
131
```

15.1 Third-party modules/libraries

We often use third-party modules/libraries such as numpy or pandas to read/write files … it's much easier. For example,

[16]:
```
import pandas as pd
df = pd.read_csv('biostats.csv')
print(df)
```

```
    Name       "Sex"   "Age"   "Height (in)"   "Weight (lbs)"
0   Alex         "M"      41              74              170
1   Bert         "M"      42              68              166
2   Carl         "M"      32              70              155
3   Dave         "M"      39              72              167
4   Elly         "F"      30              66              124
5   Fran         "F"      33              66              115
6   Gwen         "F"      26              64              121
```

```
7   Hank        "M"      30              71              158
8   Ivan        "M"      53              72              175
9   Jake        "M"      32              69              143
10  Kate        "F"      47              69              139
11  Luke        "M"      34              72              163
12  Myra        "F"      23              62               98
13  Neil        "M"      36              75              160
14  Omar        "M"      38              70              145
15  Page        "F"      31              67              135
16  Quin        "M"      29              71              176
17  Ruth        "F"      28              65              131
```

saving a csv file is also a lot easier...

[14]:
```
df.to_csv('biostats_n.csv', index=False)
```

Let's reload this new csv file

[17]:
```
df = pd.read_csv('biostats_n.csv')
print(df)
```

```
    Name       "Sex"   "Age"   "Height (in)"   "Weight (lbs)"
0   Alex         "M"      41              74              170
1   Bert         "M"      42              68              166
2   Carl         "M"      32              70              155
3   Dave         "M"      39              72              167
4   Elly         "F"      30              66              124
5   Fran         "F"      33              66              115
6   Gwen         "F"      26              64              121
7   Hank         "M"      30              71              158
8   Ivan         "M"      53              72              175
9   Jake         "M"      32              69              143
10  Kate         "F"      47              69              139
11  Luke         "M"      34              72              163
12  Myra         "F"      23              62               98
13  Neil         "M"      36              75              160
14  Omar         "M"      38              70              145
15  Page         "F"      31              67              135
16  Quin         "M"      29              71              176
17  Ruth         "F"      28              65              131
```

Summary

In this chapter, we learned about reading/writing text and csv files. Using third-party libraries such as numpy or pandas makes the reading/writing of files a lot easier.

Chapter 16

Plotting with Matplotlib

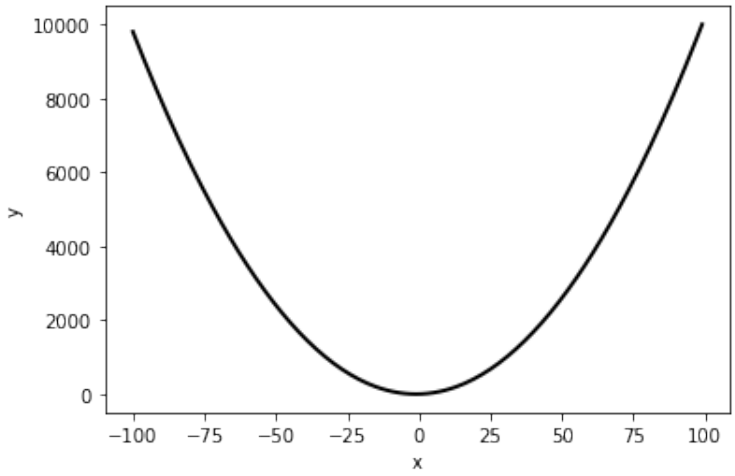
Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. Here, I'll show a few examples only. For more details and examples, you can check https://matplotlib.org/

Example 1 Plot a line graph for the polynomial $y = x^2 + 2x + 1$

[1]:
```
# create data
polynomial = lambda x: x**2 + 2*x + 1
x = list(range(-100, 100))
y = [polynomial(val) for val in x]
```

[4]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(x, y, linewidth=2, color='k')
plt.xlabel('x')
plt.ylabel('y')
```

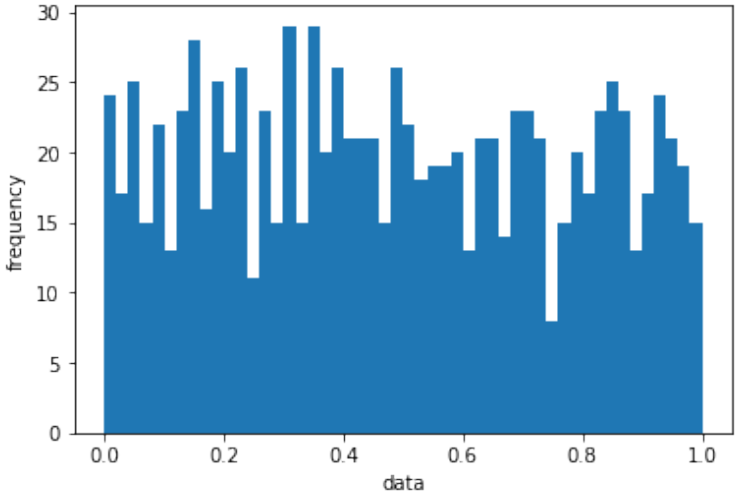
[4]:
```
Text(0, 0.5, 'y')
```



Example 2 Generate 1000 uniformly distributed random numbers between 0 and 1 and plot their histogram.

[6]:
```
# create data
from random import random
y = [random() for _ in range(1000)]
# plot histogram
plt.hist(y, bins=50, range = (0, 1))
plt.xlabel('data')
plt.ylabel('frequency')
```

[6]:
```
Text(0, 0.5, 'frequency')
```



Summary

In this chapter, we just briefly introduced Matplotlib, which is a key library for plotting and data visualization. For more examples and details, check https://matplotlib.org/

Chapter 17

Classes

Python is an object-oriented programming language. Almost everything, in python, is an object.

> **An object** is an instance of a class

We have seen earlier that type() method can be used to determine the class type of a given object. For example

[1]:
```
a = 2
print(type(a))
```

```
<class 'int'>
```

[2]:
```
def my_work():
    print('this is my work')
print(type(my_work))
```

```
<class 'function'>
```

since integers, floats, lists, functions,.. and so on, are objects in python, we can assign each of them to variables. This brings additional flexibility to python programming language. For example,

[4]:
```
f = my_work
f()
```

```
this is my work
```

[5]:
```
del my_work   #deleting my_work() function
```

[6]:
```
# now if we run my_work(), it'll generate error
my_work()
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)
```

```
    <ipython-input-6-2a7ab4279cec> in <module>
      1 # now if we run my_work(), it'll generate error
----> 2 my_work()


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

[7]:
```
#However, we can still use f()
f()
```

```
this is my work
```

> **A class** is kind of a template that allows including behavior(methods) and state(variables/data/attributes) to a particular object.

When objects are created by a class, they inherit the class attributes and methods. Let's see how we can write a simple class

[41]:
```
class Vehicle:                    # it is recommended to write class names in␣
↪upper camel case such as MyCar
    ''' This class designs a basic vehicle model'''

    def __init__(self, *args):            #initializing the class, self points␣
↪to the class itself
        self.color = args[0]              # instance variables (state, data or␣
↪attributes)
        self.doors = args[1]
        self.tires = args[2]

    def __str__(self):    # description of instance when printed
        return f'This {self.doors}-door vehicle has {self.color} color and {self.
↪tires} tires.'

    def __repr__(self):   # text you would type to re-create the instance.
        return f'{self.__class__.__name__}({self.color!r}, {self.doors!r}, {self.
↪tires!r})'
    #------------------------methods/behavior-------------------------------
    def normal_brake(self):
        return "slowing down ..."

    def hard_brake(self):
        return "screeeeeeeeeech!"

    def turn_right(self):
        return "turning right"
```

```
    def turn_left(self):
        return "turning left"

    def use_front_wipers(self):
        return "cleaning windscreen ..."

    def make_noise(self):
        pass                    #in python, pass key is used as a placeholder␣
↪for future use
```

[42]:
```
car = Vehicle('red', 4, 4)              #creating an instance of the Vehicle class
```

[43]:
```
car                         #this invokes the __repr__() method and shows the text␣
↪to recreate the instance
```

[43]:
```
Vehicle('red', 4, 4)
```

[44]:
```
print(car)                # this invokes the __str__() method
```

```
This 4-door vehicle has red color and 4 tires.
```

[45]:
```
print(car.normal_brake())    # check the behavior of the car object
```

```
slowing down ...
```

17.1 Four principles of object-oriented programming

The four principles of object-oriented programming are encapsulation, abstraction, inheritance, and polymorphism. If we read the definitions of these words/terms in an English dictionary, we can gain a basic understanding of how these principles are used in a programming language.

1. **Encapsulation:** (definition: the action of enclosing something in or as if in a capsule). In programming, it is the ability to restrict access to variables and methods, thus preventing them from being modified by accident. In python, we use a double underscore (__) at the start of method or variable's name to indicate it is private.
2. **Abstraction:** (definition: the quality of dealing with ideas rather than events). In programming, it means hiding the real complex implementations and only showing the essential features of the object, such that users only know how to use it. Python offers the abstract base class module to create abstract classes and methods.
3. **Inheritance:** As the name suggests, someone is inheriting something from someone. Inheritance allows creating new classes using classes that have already been defined. In this case, the class from which another class inherits attributes and methods is called the base class, whereas the inheriting class is called the derived class.
4. **Polymorphism:** (definition: the condition of occurring in several different forms). In programming, it is the ability to process objects differently depending on their data type or class.

Ummm.....not sure what I mean. Let's try to explain this by writing some code.

Suppose we want to design a gasoline car model. Since a car belongs to the vehicle category, it should share the same basic attributes (it should have a color, a number of tires, and doors, ...) and same behavior (it should be able to turn, stop, drive, ...). In this case, we can simply create another subclass Gasolinecar which will inherit the attributes and methods from the Vehicle class

[67]:
```
class GasolineCar(Vehicle):       # GasolineCar is the derived class whereas␣
↪Vehicle is the base class --> Inheritance
    def __init__(self, *args):    # initialize the instance of GasolineCar
        super().__init__(*args)   # initialize the parent/base class
        self.__engine = 'v6'      # private variable (should only be changed␣
↪using the class methods) --> encapsulation
    def lock_windows(self):       # Extending the base class, adding more␣
↪functionality
        return "Windows are locked."
    def make_noise(self):         # method overridding -- processing this object␣
↪differently than the base class (polymorphism)
        return "Vrrrrrooooooooom"
    def get_engine(self):         # getter method to display the private␣
↪variable (.__engine)
        print(self.__engine)
    def set_engine(self, engine): # setter method to change the private variable
        self.__engine = engine
```

[47]:
```
b = GasolineCar('red', 4, 4)      # create an instance of GasolineCar
```

[48]:
```
print(b.normal_brake())           # it inherited the methods from the base class␣
↪(Vehicle)
```

```
slowing down ...
```

[49]:
```
print(b.make_noise())             # method used specific to this class -
```

```
Vrrrrrooooooooom
```

[50]:
```
b                                 # invoking the __repr__() method - text to␣
↪recreate this instance
```

[50]:
```
GasolineCar('red', 4, 4)
```

[51]:
```
print(b)                          # invoking the __str__() method
```

```
This 4-door vehicle has red color and 4 tires.
```

Now, let's say we want to design an electric car model. For this, we can create another subclass which also inherits from the Vehicle class. You can see, how it helps the code reusability ....

[68]:
```
class ElectricCar(Vehicle):
    def __init__(self, *args):
        super().__init__(*args)
        self.__engine = 'motor'
    def make_noise(self): #same method as used in GasolineCar class, but␣
 ↪different function specific to this class ...
        return "I can only make artificial noise under certain conditions ..." ␣
 ↪# ... (an example of polymorphism)
    def get_engine(self):
        print(self.__engine)
    def set_engine(self, engine):
        self.__engine = engine
```

[69]:
```
c = ElectricCar('red', 4, 4)
```

[70]:
```
print(c.hard_brake())
```

```
screeeeeeeeeech!
```

[71]:
```
print(c.make_noise())
```

```
I can only make artificial noise under certain conditions ...
```

[72]:
```
print(c.__engine)
```

```
---------------------------------------------------------------------------

AttributeError                            Traceback (most recent call last)

<ipython-input-72-7d6935b5390f> in <module>
----> 1 print(c.__engine)


AttributeError: 'ElectricCar' object has no attribute '__engine'
```

[73]:
```
# but we can access this private variable using the class method
c.get_engine()
```

```
motor
```

[74]:
```
# we can change this variable as well using the set method from the same class
c.set_engine("new motor")
c.get_engine()
```

```
new motor
```

Summary

In this chapter, we learned about objects, classes and important features of object-oriented programming (inheritance, polymorphism, encapsulation, abstraction).

Chapter 18

Summary

This is all from this python programming tutorial, in which we covered the following topics:

1. Introduction on python programming
2. Comments and print()
3. Data types/structures
4. Strings
5. Operators
6. Lists and tuples
7. Dictionaries
8. Sets
9. if-elif-else (conditional programming)
10. Loops (while and for constructs)
11. Comprehensions
12. Functions
13. Import modules
14. Decorators
15. Read/write text or csv files
16. Plotting with Matplotlib
17. Classes

I hope the material in this tutorial helps in writing python programs more effectively. Off-course python is much more than this. However, these concepts will be applicable/helpful whatever new package/library you use in your python programming assignment/project.

Let's end this tutorial by writing the code for a guessing game, in which a user has to guess a randomly generated number between 0 and 10 in 4 attempts.

[14]:
```
from random import randint
def guess_this_number(max_attempts=4):
    '''Guess a random number(integer) between 0 and 10 in 4 attempts'''
    number=randint(0, 10)
    guess = int(input('enter a number between 0 and 10: '))
    for count in range(max_attempts):
        if count < max_attempts-1:
            if guess < number:
                guess = int(input(f'guess # {count+1} = {guess} is smaller,␣
 ↪please enter a bigger number: '))
            elif guess > number:
                guess = int(input(f'guess # {count+1} = {guess} is bigger,␣
 ↪please enter a smaller number: '))
            else:
                print(f'Congratulations! you guessed it right in {count+1}␣
 ↪attempts')
                break
        else:
            if guess == number:
                print(f'Congratulations! you guessed it right in {count+1}␣
 ↪attempts')
                break
    else:
        print(f'Sorry, play again! The correct number was {number}')
```

[19]:
```
guess_this_number()
```

```
enter a number between 0 and 10: 5
guess # 1 = 5 is smaller, please enter a bigger number: 7
guess # 2 = 7 is smaller, please enter a bigger number: 9
guess # 3 = 9 is bigger, please enter a smaller number: 8
Congratulations! you guessed it right in 4 attempts
```