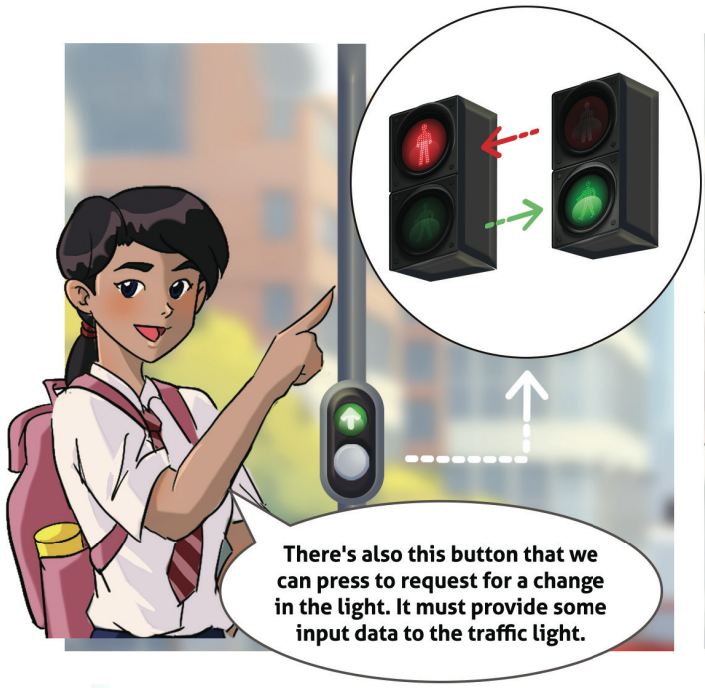
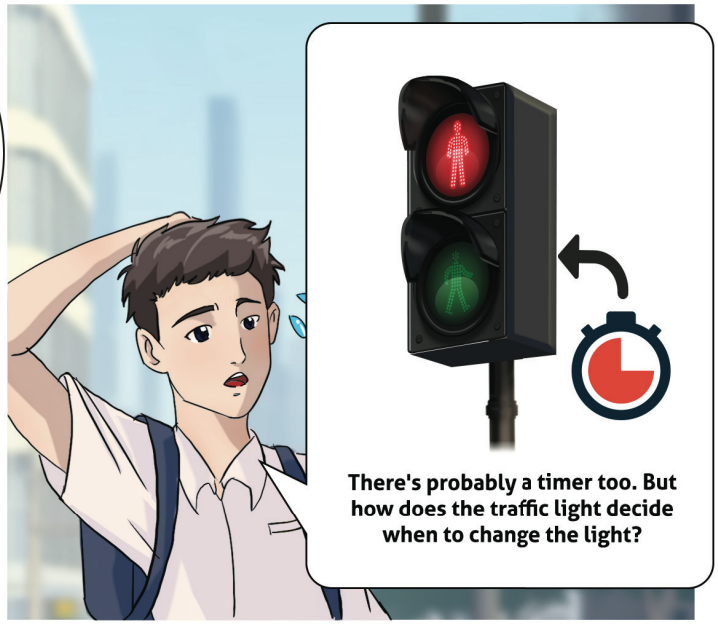
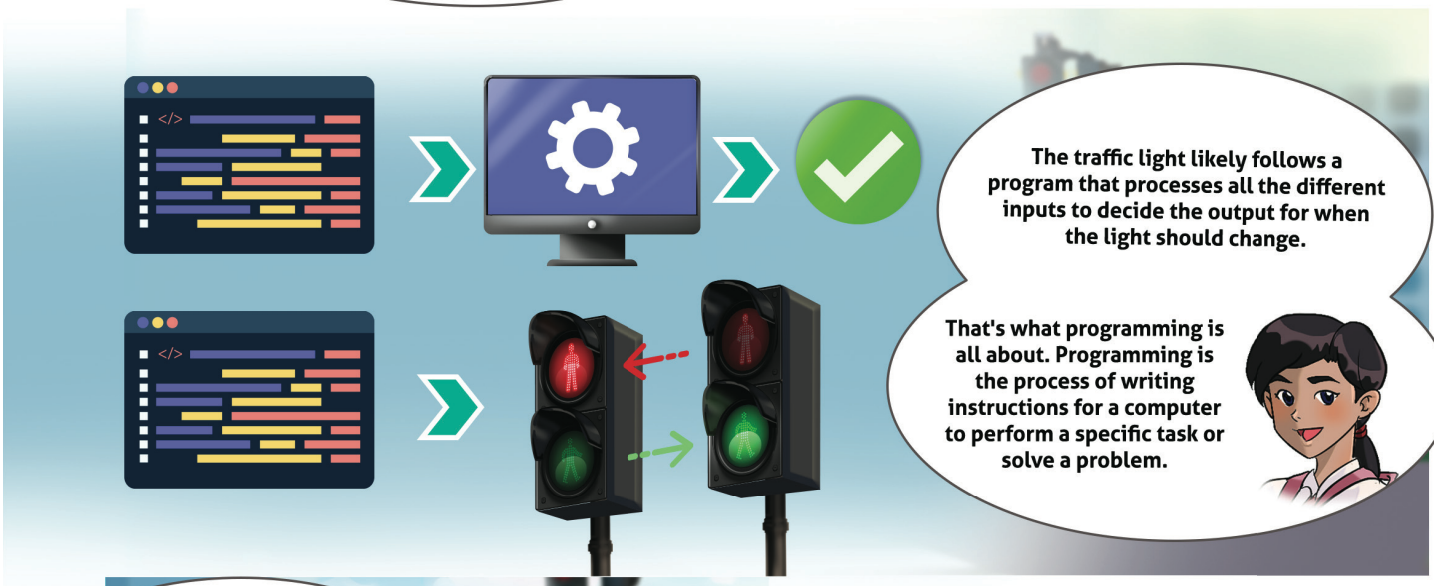
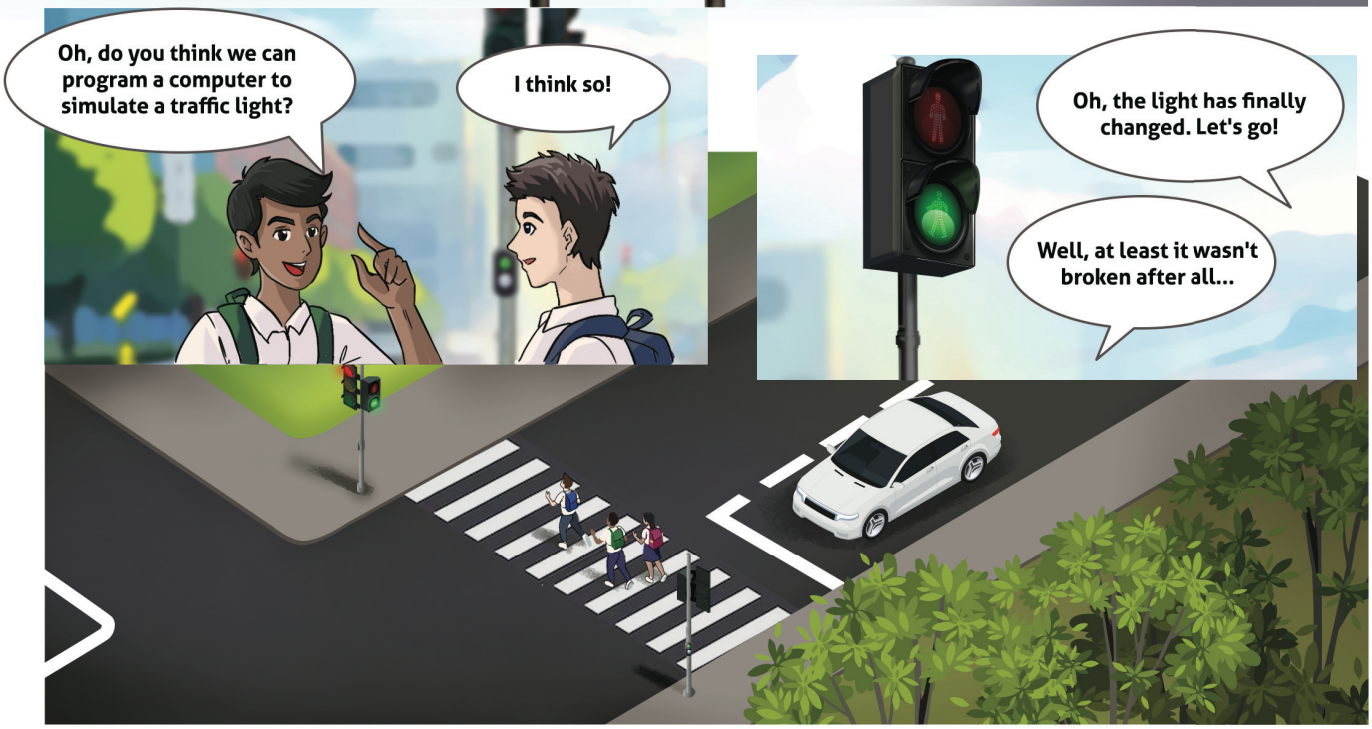
CHAPTER 04

Programming


There's also this button that we can press to request for a change in the light. It must provide some input data to the traffic light.


There's probably a timer too. But how does the traffic light decide when to change the light?


The traffic light likely follows a program that processes all the different inputs to decide the output for when the light should change.
That's what programming is all about. Programming is the process of writing instructions for a computer to perform a specific task or solve a problem.


Oh, do you think we can program a computer to simulate a traffic light?
I think so!
Oh, the light has finally changed. Let's go!
Well, at least it wasn't broken after all...

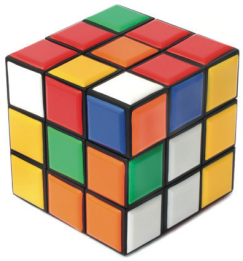
4.1 Introduction to Algorithms and Programming

To process data, computers follow a set of instructions. In this chapter, we will learn about how such instructions may be provided using algorithms and programming.

4.1.1 Algorithms

An **algorithm** is set-by-step instructions for solving a problem or performing a calculation.

For instance, the step-by-step processes of solving a Rubik's cube or a Sudoku puzzle can be considered algorithms.



Rubik's cube

8		6			3		9	
	4			1			6	8
2			8	7				5
1		8			5		2	
	3		1				5	
7		5		3		9		
	2	1			7		4	
6				2		8		
	8	7	6		4			3

Sudoku puzzle

Figure 4.1 Examples of games and applications involving algorithms

KEY TERMS

Algorithm
A set of step-by-step instructions for solving a problem or performing a calculation

4.1.1.1 Example: Addition Algorithm

We use algorithms every day, although we may not see them explicitly written out. For example, the following steps can be used to add two positive whole numbers:

Step 1: Write the first number, then write the second number below it such that the digits in the ones place are aligned vertically. Draw a horizontal line below the second number, as in Figure 4.2.

First number 2017
Second number 1965

Figure 4.2 Step 1 of addition algorithm

Step 2: Let the current column be the right-most column of digits in the ones place.

Step 3: Add all the digits (including any "carry-over" digits) in the current column.
If a digit is missing, treat it as 0. There is no "carry-over" digit for the right-most column.

Step 4: If the result is greater than 9, write "1" at the top of the column immediately to the left of the current column as a "carry-over" digit.

Step 5: Write the last digit of the result at the bottom of the current column, below the line.

Step 6: If there are still digits (including "carry-over" digits) to the left of the current column, redefine that column on the left as the new current column and go back to Step 3. Otherwise, the final answer is formed by the row of digits below the line.

"Carry-over" digits		1		
First number	2	0	1	7
Second number	1	9	6	5
Final answer	3	9	8	2

Figure 4.3 Example of completed addition algorithm

The advantage of writing out an algorithm step by step is that these steps can be followed precisely by anyone who can add digits to obtain the correct answer. This algorithm is also useful because it is general and works for adding any two positive whole numbers.

4.1.1.2 Example: Algorithm for Calculating the Mean Subject Grade for Seven Subjects

In some secondary schools, the Mean Subject Grade (MSG) is used to measure a student's performance in school. The MSG is a number between 1 (best) and 9 (worst) calculated from the student's rounded percentage scores in each subject.

The following steps provide an algorithm to calculate the MSG of a student taking seven subjects:

Step 1: Convert the rounded percentage scores into grade points using the table to the right.

Step 2: Add up all the grade points.

Step 3: Divide the result by 7.

Step 4: Round the result to two decimal places. This is the student's MSG.

Rounded percentage score	Grade	Grade points
75–100	A1	1
70–74	A2	2
65–69	B3	3
60–64	B4	4
55–59	C5	5
50–54	C6	6
45–49	D7	7
40–44	E8	8
Less than 40	F9	9

Table 4.1 Converting rounded percentage scores to grade points

For instance, if the rounded percentage scores for the student's seven subjects are as shown in the illustration below, the corresponding MSG would be 3.14.

This algorithm is useful as anyone can follow the steps precisely to obtain the MSG of any student (as long as the student takes exactly seven subjects).

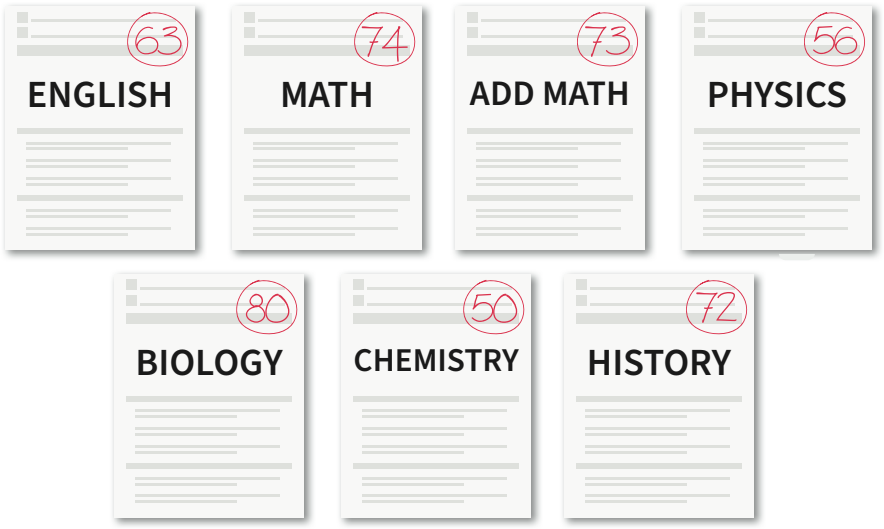


Figure 4.4 Rounded percentage scores of a student taking seven subjects

4.1.2 Programs

An algorithm describes the steps necessary to solve a problem, but for a computer to perform these steps, the instructions need to be written for a computer to understand.

When instructions are written in a form that can be run on a computer, we call such instructions a **program**. In general, programs must be written using a programming language and process of writing programs is called programming.

Instructions written in a programming language are called **source code**, source, or even just code for short. The programming language we will learn in this textbook is called Python and the following is an example of Python source code:

```
total = test_score_1 + test_score_2
```

Figure 4.5 Example of Python source code

Besides Python, there are a variety of other programming languages. Figure 4.6 shows some examples of source code for the same instruction in other programming languages such as C and Scratch.

KEY TERMS

Program
A set of instructions written to perform specific tasks on a computer

Source code
Instructions written in a programming language

Python logo

C

```
total = test_score_1 + test_score_2;
```

Pascal

```
total := test_score_1 + test_score_2;
```

R

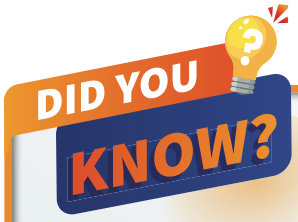
```
total <- test_score_1 + test_score_2
```

Scratch

Lisp

```
(defvar *total* (+ *test_score_1* *test_score_2*))
```

Figure 4.6 Examples of source code in other programming languages



In general, a computer's processor cannot run source code directly. Instead, processors require instructions to be provided in a format called "machine code" that is difficult for humans to read. In practice, source code must first be translated to machine code before it can be run on a computer.

1. The following is an algorithm that can be performed on any digit from 1 to 9.

Step 1: Multiply the input digit by 9.
Step 2: Multiply the result by 12345679. This is the output.

Determine the algorithm's output when the input digit is:
a) 1
b) 2
c) 7
d) 9

2. The following is an algorithm that can be performed on any positive whole number. (Make sure to use the denary number system.)

Step 1: Calculate the difference between the input number and the input number with its digits reversed.
Step 2: Add up all the digits of the result (ignore the sign).
Step 3: If the result has only 1 digit, this is the output. Otherwise, go back to Step 2.

Determine the algorithm's output when the input number is:
a) 21
b) 382
c) 9531
d) 40004

QUICK CHECK 4.1

3. The following is an algorithm used for decode secret messages. It requires a secret message and a positive whole number to be provided as inputs.

Step 1: Extract the first N letters of the secret message where N is the provided number.
Step 2: Append the extracted letters to the remaining letters of the secret message. This is the output.

For example, if the inputs are:
Secret message : MPLEEXA
Number : 4

For Step 1, the extracted 4 letters are MPLE. For Step 2, the extracted letters MPLE are appended to the remaining letters EXA to obtain the output EXAMPLE.

Now, determine the algorithm's output when the inputs are:
a) Secret message: ENTSIL
Number: 3
b) Secret message: EDACTR
Number: 5
c) Secret message: EBY
Number: 1

4. State whether each of following statements are true or false.
a) Python is a programming language.
b) Python is an algorithm.
c) A processor is a made of instructions.
d) A program is a made of instructions.

4.2 Defining Problems

LEARNING OUTCOMES

2.1.1 *For a given problem, identify and remove unnecessary details to specify the*
- *inputs and the requirements for valid inputs*
- *outputs and the requirements for correct outputs*

Algorithms can be used to solve problems. To define a problem, we need to specify its:

1. Inputs and requirements for valid inputs
2. Outputs and requirements for correct outputs

Here, **output** is the result that the algorithm produces while **input** is any data that can affect what we require for the output.

An algorithm describes the process of transforming inputs into outputs. A **solution** is defined as an algorithm that solves the given problem by always giving correct outputs when valid inputs are provided.

KEY TERMS

Input (algorithms)
Data used by an algorithm that can affect what is required for correct output

Output (algorithms)
Results produced by an algorithm

Solution
An algorithm that always gives correct outputs when provided with valid inputs

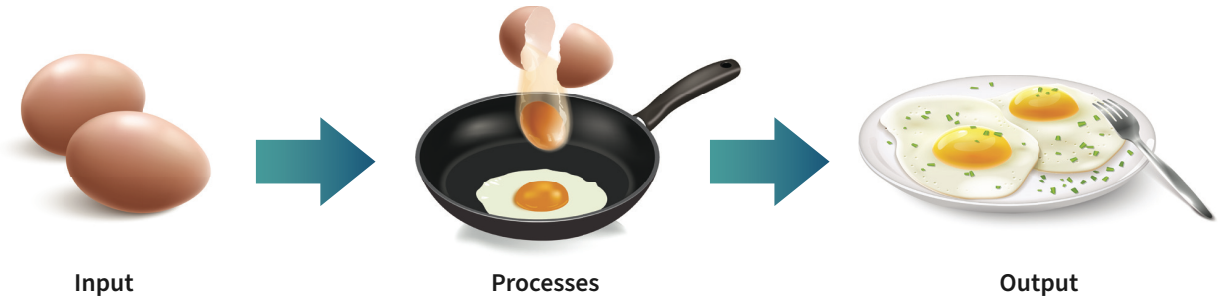


Figure 4.7 Process flow in problem-solving

4.2.1 Specifying the Inputs

It is not always easy to specify the inputs for a problem correctly. Good input specifications must:

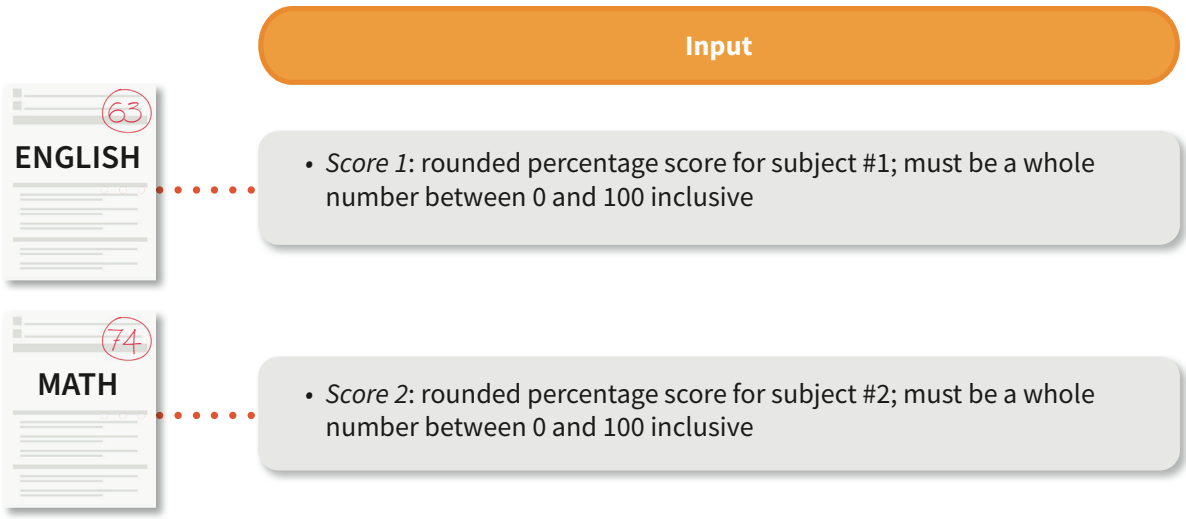
1. include only the important data that can affect what we require for the output and exclude any irrelevant details; and
2. state the range of valid or acceptable values for these inputs.

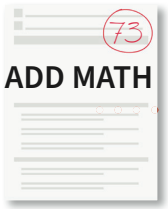
It is also a good practice to give each input a name. For instance, Table 4.2 shows how we can specify the inputs to the addition problem that can be solved using the algorithm in section 4.1.1.1:

Input
• *x*: a positive whole number • *y*: a positive whole number

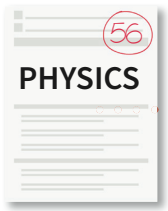
Table 4.2 Specifying the inputs to the addition problem for whole numbers (positive only)

Similarly, the inputs to the problem of calculating the MSG for a student taking seven subjects can be specified by treating each subject's rounded percentage score as a separate input:

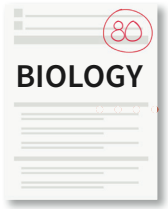




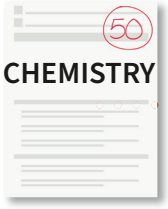
- *Score 3*: rounded percentage score for subject #3; must be a whole number between 0 and 100 inclusive



- *Score 4*: rounded percentage score for subject #4; must be a whole number between 0 and 100 inclusive



- *Score 5*: rounded percentage score for subject #5; must be a whole number between 0 and 100 inclusive

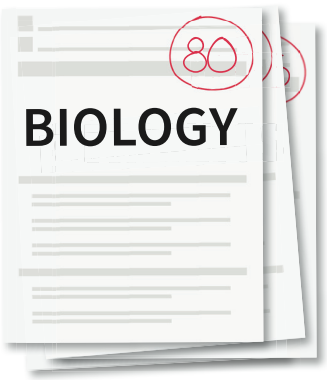


- *Score 6*: rounded percentage score for subject #6; must be a whole number between 0 and 100 inclusive

- *Score 7*: rounded percentage score for subject #7; must be a whole number between 0 and 100 inclusive

Table 4.3 Specifying the inputs to the problem of calculating the MSG for seven subjects

However, this way of specifying the inputs can be quite repetitive. Since the requirements for each rounded percentage score are the same, we can shorten the specification as follows:



Input

- *Score*: rounded percentage score for subject; must be a whole number between 0 and 100 inclusive (provided seven times, once for each subject)

Table 4.4 Simplified way of specifying the inputs to the problem of calculating the MSG for seven subjects

4.2.2 Specifying the Outputs

To specify the outputs for a problem correctly, we need to include all the important features that the output is required to have. Any details that are not mentioned are assumed to be unimportant or irrelevant, and thus these details may differ from the output of one solution to another.

For instance, Table 4.5 shows how we can specify the output of the addition problem that can be solved using the algorithm in section 4.1.1.1.

OUTPUT
• *The sum of x and y*

Table 4.5 Specifying the output of the addition problem for whole numbers (positive only)

Similarly, we can specify the output of the problem of calculating the MSG for a student taking seven subjects in this manner:

OUTPUT
• *MSG for the seven subjects*

Table 4.6 Specifying the output of the problem of calculating the MSG for seven subjects (original)

However, if we require the MSG values to be rounded to two decimal places, we can revise the problem of calculating the MSG for seven subjects accordingly:

OUTPUT
• *MSG for the seven subjects, rounded to two decimal places*

Table 4.7 Specifying the output of the problem of calculating the MSG for seven subjects (rounded)

1. Modify the input specifications for the addition problem in Table 4.2 so that both positive and negative whole numbers are valid inputs. Determine whether the original addition algorithm is a solution to this modified problem, and if not, explain why.

2. Alex wants to find the weight of the heaviest apple on display at the market.
Specify the input and output requirements for Alex's problem.

3. Table 4.8 shows a possible set of input and output requirements for calculating the average score of a class test. Suggest how it can be improved and why.

Table 4.8 Input and output requirements for the average-score problem

Input	Output
• *Title*: name of the class test • *Scores*: list of class test scores with each student's score included once	• The average score of the class test

4.3 Installing Python

The programming language that you will learn in this textbook is Python. It can be downloaded for free online.

There are two ways to use Python:

1. **Graphical user interface**
   Python can be used with programs such as JupyterLab Desktop with menus and **syntax highlighting** to facilitate the writing and running of Python source code. Many alternatives to JupyterLab Desktop are also available for download online. Such graphical user interfaces are most useful if the programmer wishes to use an "all-in-one" program to edit, run and test his or her source code.

2. **Command line interface**
   Python can also be run using a command line interface where commands are given as lines of text. This interface is most useful for combining the use of Python with other programs or for automating the use of Phyton. In fact, graphical user interfaces for Python typically make use of the command line interface in the background.

KEY TERMS

Command line interface
A means of interacting with a program such that commands are given as lines of text

Graphical user interface
A means of interacting with a program such that commands are given using visual elements such as windows, icons, menus and mouse pointers

Syntax highlighting
Displaying different parts of source code differently (such as using different colours and fonts) in a text editor based on the rules of the programming language to make reading easier

4.3.1 Graphical User Interface

Figure 4.8 shows what JupyterLab Desktop looks like when it is initially run on Windows:

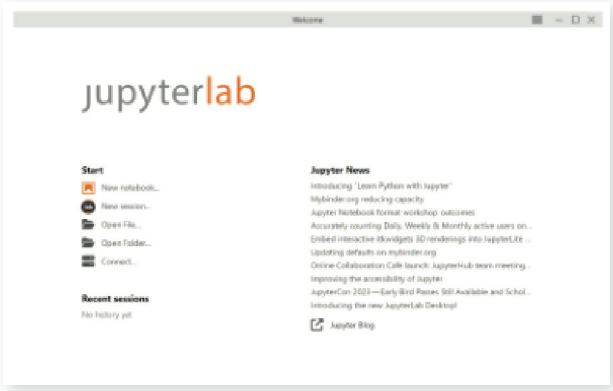


Figure 4.8 JupyterLab Desktop on initial start-up

Unlike traditional programming tools, programs in JupyterLab Desktop are written in files called **notebooks** where executable code is embedded in formatted documents that may contain text, images and other media. Figure 4.9 shows JupyterLab Desktop editing a notebook after the "New Notebook…" option is selected.

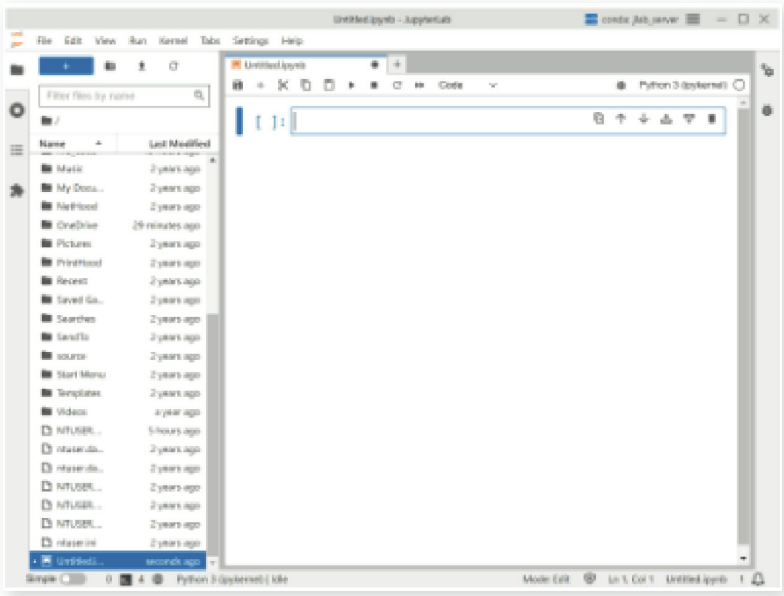


Figure 4.9 Editing a notebook

By default, selecting "New Notebook…" creates a new file named `Untitled.ipynb`. You can rename it by right-clicking on the notebook's tab at the top or in the file browser on the left. You can also create a new notebook using the File menu or the plus sign on the tab bar. However, if you create a new notebook in this way, you will need to select a **kernel** that runs the code you write. For this textbook, any available Python kernel can be used.

KEY TERMS

Kernel (Jupyter)
A program that runs the code embedded in a notebook

Notebook (Jupyter)
A file where executable code is embedded in formatted documents that may contain text, images and other media

Each notebook is made up of cells, and each cell may contain either executable code or formatted content (e.g., text, images, media). The first cell in a new notebook is usually a code cell where you can enter source code. For instance, try entering the following line of Python source code in the cell and pressing Shift-Enter:

```
print("Hello, World!")
```

Figure 4.10 Entering source code into a code cell

Note that for the code examples given in this textbook, anything that you may need to type will be shown in a cell with a gray background. To run the code, you will usually need to press Shift-Enter or click the "play" button on the toolbar.

If there are no errors, you should see the phrase “Hello, World!” displayed on the screen immediately below the code cell:

```
print("Hello, World!")
Hello, World!
```

Figure 4.11 Output of a code cell

If there is an error, Python will display an error message, such as the example in Figure 4.12.

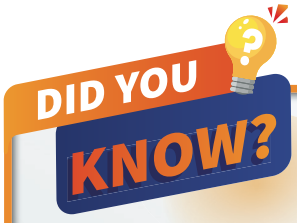
```
Print("Hello, World!")

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[2], line 1
----> 1 Print("Hello, World!")

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

Figure 4.12 Example of an error message being displayed

Note that the Python programming language is case-sensitive and that “Print” (with an upper-case “P”) is not the same as “print” (with a lower-case “p”), as shown by the error in Figure 4.12. It is typical to encounter such errors when learning a new programming language. You will learn more in Chapter 6 on what to do if a program does not work as intended.

You may occasionally run Python code that that “hangs” JupyterLab so it refuses to run any other code. If this happens, select “Restart Kernel…” from the Kernel menu and click on “Restart” to confirm.

Some programs in this textbook take up multiple lines. When entering such programs, take note of the difference between pressing Enter and pressing Shift-Enter. Pressing Enter just inserts a new line without running the code. To run the code in a code cell, you need to press Shift-Enter.



JupyterLab Desktop is an example of an “integrated development environment” (IDE) that makes it easier to use Python by providing an “all-in-one” program for editing, running and testing source code.

Notebooks in Jupyter Desktop are stored in `ipynb` format. The `ipynb` extension stands for “Interactive Python Notebook” and notebooks stored in this format are compatible with web-based tools such as Google Colab, Anaconda Cloud and JupyterLite that are freely available to use online.

Python programs may also be written in plain text files with the `py` extension. To run such programs in Jupyter Desktop, you can either copy the `py` file’s contents into a code cell of a notebook and press Shift-Enter, or enter the following “magic” command in a cell and press Shift-Enter, replacing “example.py” with the name of the py file:

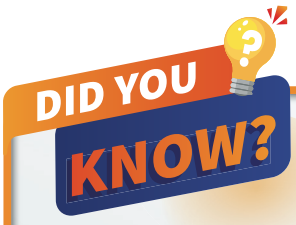
```
%run example.py
```

Figure 4.13 Running a Python program stored in a py file

JupyterLab Desktop also has an option to show line numbers under the View menu. The following is an example of a Python program with multiple lines of code:

```
# Program 4.1: hello_world.ipynb

1 print ("Hello, World")
2 print ("This program prints two lines of output.")

Hello, World
This program prints two lines of output.
```

Figure 4.14 Program with multiple lines of code



Instead of navigating menus to reach commands, you can activate JupyterLab Desktop's "command palette" by pressing Ctrl-Shift-C to search for commands instead. For example, by pressing Ctrl-Shift-C and typing "line", you can toggle line numbers without needing to memorise where "Show line numbers" is found in the menus.

JupyterLab Desktop will also attempt to auto-complete any partially written code when you press the Tab key. This can be convenient for long names or when you cannot recall the exact name of something in Python.

4.3.2 Command Line Interface

While JupyterLab Desktop can be convenient, a graphical user interface usually requires the use of a mouse, making tasks difficult to automate without human intervention. A command line interface, on the other hand, can be automated easily as commands are given using only lines of text. Use of Python's command line interface, however, is not covered in this textbook.

4.4 Comments

When you save a Python program in a file, you should also add **comments** to your program. Comments make the source code easier for other people to understand but will be ignored by the Python interpreter. In Python, comments start with the hash symbol (#) and finish when the line ends. Any text from the hash symbol until the end of the line will be ignored by the interpreter. For example, Figure 4.15 shows `hello_world.ipynb` with some additional comments:

```
Program 4.2: hello_world_with_comments.ipynb

# This is a comment at the start of a program. Usually, comments at
# the start of a program explain the purpose of a program or
# describe its expected inputs and outputs.

print("Hello, World!") # This is also a comment.
print("This program prints two lines of output.")
```

Figure 4.15 `hello_world.ipynb` with added comments

Even with the addition of comments, this program still behaves exactly like the original program, as shown in Figure 4.16:

```
# This is a comment at the start of a program. Usually, comments at
# the start of a program explain the purpose of a program or
# describe its expected inputs and outputs.

print("Hello, World!") # This is also a comment.
print("This program prints two lines of output.")
```

```
Hello, World!
This program prints two lines of output.
```

Figure 4.16 Output in Jupyter Desktop

KEY TERMS

Comment
A piece of source code that explains the program but is ignored by Python

4.5 Literals and Variables

LEARNING OUTCOMES

2.3.1 *Use variables to store and retrieve values.*

2.3.2 *Use literals to represent values directly in code without using a variable.*

Computers are designed to store and read values quickly and reliably. To work with values in Python, we use literals and variables.

4.5.1 Literals

KEY TERMS

Literal
A value that is represented directly in source code

Syntax
Rules that determine how the words and symbols in a valid instruction are arranged

A **literal** is a value that is represented directly in source code. For instance, the number 2024 can be represented by writing out its digits in source code:

```
2024
```

Figure 4.17 Example of a literal

Different types of values have different rules or **syntax** for how literals must be written. For instance, text literals must be enclosed in either single or double quotes:

```
"This is also a literal."
```

Figure 4.18 Another example of a literal

4.5.2 Variables

Literals are useful for representing values that are relatively short and do not need to be changed by users of the program. Literal values also do not have names, so their purpose may not be obvious and they cannot be easily reused or manipulated by other parts of the program.

In contrast, variables allow us to associate meaningful names to values. Literals are often used to provide the initial values for variables, after which they can be changed using the variable while the program is running.

A variable is **initialised** when a value is first stored in it. It is usually a good practice to initialise a variable as soon as there is a value to be assigned to it.

4.5.2.1 Assigning Values to Variables

To create a variable or change the value assigned to a variable, we need a name as well as a corresponding value to be assigned. The format of the Python instruction for assigning values is the name we intend to use, followed by the equals sign (=), followed by the value to be assigned. This type of code that instructs the computer to perform a specific action is called a **statement**. The syntax of an assignment statement is summarised in Figure 4.19.

Syntax 4.1 Assignment Statement

```
variable_name = value
```

Figure 4.19 Syntax for assignment statements

Variable names (known as **identifiers**) must also follow certain rules:

- Can contain the upper-case and lower-case letters A through Z and the underscore (_)
- Can contain the digits 0 through 9 but not as the first character
- Cannot contain spaces or special symbols (e.g., “!”, “@”, “#”, “$”)
- Cannot be any of the reserved words (or **keywords**) in Figure 4.20 that have special meanings in Python
- Are case-sensitive

False	None	True	and	as
assert	break	class	continue	def
del	elif	else	except	finally
for	from	global	if	import
in	is	lambda	nonlocal	not
or	pass	raise	return	try
while	with	yield		

Figure 4.20 Reserved words with special meanings in Python

KEY TERMS

Identifier
A sequence of characters that follows certain rules and can be used as a variable name

Initialise
To store or assign a value to a variable for the first time

Keyword
A word that has a special meaning in a programming language and cannot be used as an identifier

Statement
Code that instructs the computer to perform a specific action

Valid identifiers	Invalid identifiers	Explanation
`sum_of_scores`	`sum-of-scores`	Identifiers cannot contain the hyphen character "-"
`my_class`	`class`	`class` is a reserved word
`BOX_SIZE`	`size_correct?`	Identifiers cannot contain the question mark character "?"
`test_score_3`	`3rd_test_score`	The first character of an identifier cannot be any of the digits "0" to "9"
`oneword`	`two words`	Identifiers cannot contain any spaces

Table 4.11 Examples of valid and invalid identifiers

Typically, identifiers in Python are written in lower case with words separated by underscores. To make our code easier to read, we should follow the same naming style.

For example, the code in Figure 4.21 assigns the value 2024 to a variable with the name `year`:

```
year = 2024
```

Figure 4.21 Example of an assignment statement

We can visualise what this does by imagining a sticky note with label "year" pasted on or attached to the location or address where 2024 is located in memory:

Addresses	Memory contents
24	2027
32	2026
40 (year)	**2024**
48	-12
56	2025
64	1965
72	"Hello, World!"

Figure 4.22 Assigning 2024 to the variable `year`

DID YOU KNOW?

It can sometimes be easier to imagine variables as boxes containing the values that are assigned to them, as shown in Figure 4.23:

Figure 4.23 Imagining a variable as a labelled box

However, you should be aware that this metaphor is not accurate and that some of Python's behaviour is best explained by thinking of variables as sticky notes attached to memory addresses.

In most versions of Python, you can get the memory address that a variable is attached to by using the `id()` function, as shown in Figure 4.24:

```
print(id(year))
```
```
46547440
```

Figure 4.24 Example of using the `id()` function

Of course, the memory address that is printed out will vary from computer to computer depending on which memory addresses are available to Python. Do not be alarmed if your computer does not produce the same output as your friend's!

We call the name, `year`, a variable because we can change (i.e., vary) the value that is assigned to it while the program is running.

```
year = 2024
year = 2025
year = 2026
year = 2027
```

Figure 4.24 Changing the value assigned to the variable `year`

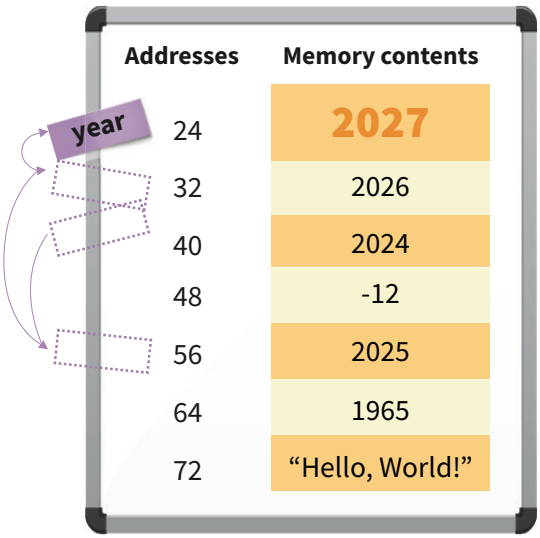


Figure 4.25 Changing the assigned value of the variable `year` from 2024 to 2027

Note that when one variable is assigned to another variable, both variables become attached to the same address. You can imagine this as moving the sticky note for one variable onto the same location as another variable. For instance, let us assign different values to two variables:

```
year_1 = 2024
year_2 = 1965
```

Figure 4.27 Assigning values to `year_1` and `year_2`

KEY TERMS

Variable
A name with an associated value that can be changed while the program is running

Expression
Code that produces a value

After Python runs the source code in Figure 4.24, the variable `year` is now assigned the value of 2027. The previous values of 2024, 2025 and 2026 are no longer associated with `year`. Only the last value that is assigned to a variable is kept.

In assigning values, what comes after the equals sign is not limited to a literal number. It can include calculations or even other variables. This type of code that produces a value is called an **expression**. For example, the following code creates a new value (the sum of 1 and 2) and attaches a variable named `total` to its address:

```
total = 1 + 2
```

Figure 4.26 Assigning the result of a calculation to the variable `total`

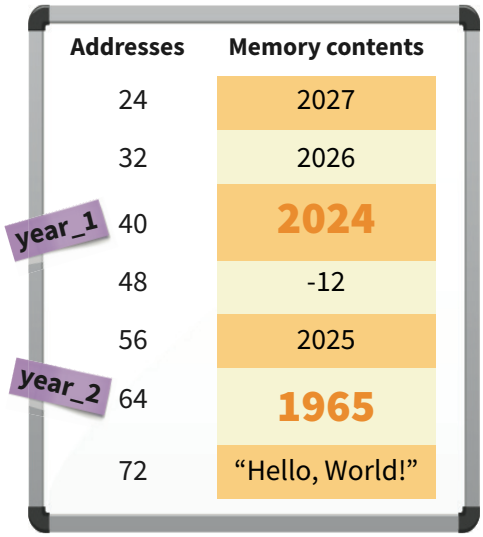


Figure 4.28 Assigning values to `year_1` and `year_2`

If we take `year_1` and assign it to `year_2`, we can imagine the sticky note for `year_2` being moved over to the location of the sticky note for `year_1`, so both `year_1` and `year_2` now refer to the same value. This is illustrated in Figure 4.29 and Figure 4.30:

```
year_2 = year_1
```

Figure 4.29 `year_1` and `year_2` now refer to the same value

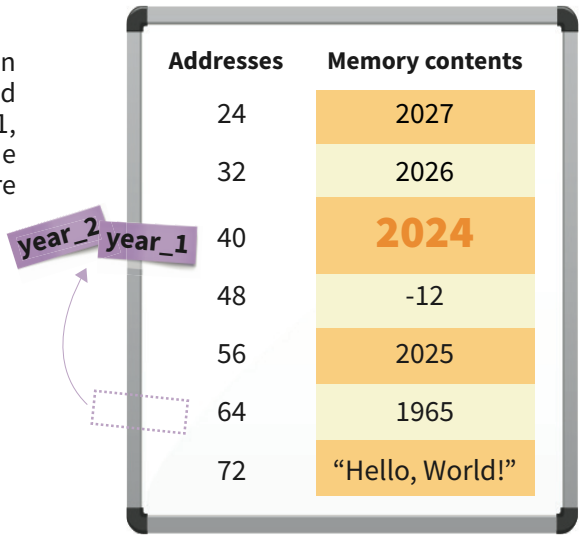


Figure 4.30 `year_1` and `year_2` now refer to the same value

4.5.2.2 Reading Values

We can retrieve or “read” the value assigned to a variable by using the variable’s name. If a value was previously assigned to a variable with the given name, Python will use the value that was assigned in place of the variable name.

To display the value of a variable on the screen, use the `print()` function as follows:

```
print(variable)
```

Figure 4.31 Displaying the value assigned to a variable

For instance, running `print(year)` would print out the value assigned to the variable named year:

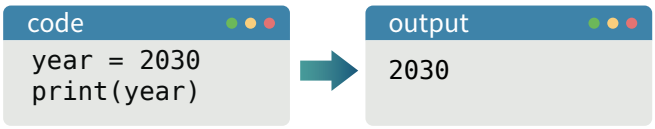


Figure 4.32 Displaying the value assigned to the variable year

However, if the name `year` does not correspond to any variable that was used or defined previously, Python will produce an error message instead:

```
print(year)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 1
----> 1 print(year)

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

Figure 4.33 Displaying the value of an undefined variable

4.5.2.3 Assigning and Reading Values Together

As mentioned previously, we can take the value from one variable, manipulate it, and then assign it to another variable. For example, the variable `next_year` will be assigned a value of 2025 after the source code in Figure 4.34 is run:

CODE:

```
year = 2024
next_year = year + 1
print(year)
```

OUTPUT:

```
2024
```

```
print(next_year)
```

```
2025
```

Figure 4.34 Using the value of `year` to assign a value to `next_year`

We can also take the value from a variable, manipulate it, and then assign it back to the same variable. This is because Python will always fully calculate the value of the right-hand side before assigning it to the variable on the left-hand side:

CODE:

```
year = 2024
year = year + 1
print(year)
```

OUTPUT:

```
2025
```

Figure 4.35 Using the value of `year` to assign a new value back to `year`

4.5.2.4 None Value

Sometimes we need to keep track of the fact that a value is missing or that there is no suitable value to be assigned to a variable. For instance, some problems may have optional inputs. Since Python will produce an error message if we try to retrieve a variable that has not been defined, we may still wish to create the variable but assign it a value that indicates the value is missing. For such cases, Python provides a special None value that can be used instead:

CODE:

```
optional_input = None
print(optional_input)
```

OUTPUT:

```
None
```

Figure 4.36 Using None to indicate a missing value

4.5.2.5 Deleting Variables

Instead of using None, it is occasionally useful to delete a variable completely so that using the variable name produces an error. This can be done using the del keyword like this:

```
year = 2024
print(year)
```

```
2024
```

```
del year
print(year)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[3], line 2
      1 del year
----> 2 print(year)

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

Figure 4.37 Deleting a variable

Syntax 4.2 del Statement

```
del variable_name
```

Figure 4.38 Syntax for del statements

1. Examine the following program:

```
hours = 6
hours = hours + 12
print(hours)
```

When Python runs the program, it performs the following steps, but not in the order presented:

A) Displays the value assigned to hours on the screen
B) Calculates the sum of hours and 12 to obtain 6 + 12 = 18
C) Assigns 6 to hours
D) Assigns 18 to hours

Arrange the steps A, B, C and D in the correct order as performed by Python.

QUICK CHECK 4.5

2. For each of the following programs, identify all the variables and predict the output:

a)
```
num_people = 6
num_chairs = 3
shortage = num_people - num_chairs
print(shortage)
```

b)
```
week_length = 7
today = 4
today = today + week_length
print(today)
```

c)
```
cost = 10
cost = cost + cost
savings = 250
savings = savings - cost
print(savings)
```

4.6 Functions, Methods and Operators

4.6.1 Functions

Just as variables can store values, **functions** can be thought of as a way of storing instructions.

You have already seen functions used in some previous examples. For example, in Figure 4.39, the `print()` function takes in a value, called an **argument**, and displays it on the screen without quotes around the content.

When using a function, type its name, followed by parentheses that can either be empty or contain comma-separated arguments (as shown in Figure 4.40). When a function is used, Python will assign any arguments to new variables before running the instructions stored in the function. This process is also known as a **function call**.

```
print("Hello, World!")
Hello, World!
```

Figure 4.39 Using a function

Syntax 4.3 Function Call

```
function_name()
function_name(argument_1)
function_name(argument_1, argument_2)
```

Figure 4.40 Syntax for function calls

KEY TERMS

Argument
An additional value that can be supplied as input to a function call

Function
A set of instructions assigned to a name that can be used again later

Function call
The process of assigning arguments to new variables and running the instructions assigned to a function

DID YOU KNOW?

You may be familiar with "argument" to mean "having a verbal disagreement" or "reasons for supporting an idea", but in mathematics the word also means "quantity from which another quantity can be deduced or calculated". The use of "argument" here is based on this mathematical definition.

Figure 4.41 shows some examples of function calls using the `print()` function.

This syntax is why a set of parentheses are used after function names to distinguish them from variable names.

In addition to the `print()` function, Python comes with many other built-in functions that you can use straight away. Some functions can even **return values** for your program to assign or use in other ways. For instance, the `max()` function accepts multiple numbers and returns the largest number:

```
result = max(3, -2, 1, 5, 0)
print(result)
```

```
5
```

Figure 4.42 Using the `max()` function

In this example, the return value of `max()` is the largest argument 5. Hence, the entire function call "`max(3, -2, 1, 5, 0)`" is treated as the number 5, and the number 5 is in turn assigned to the variable `result`.

The return value (or result) of one function call can be used as an argument for another function call. This is called a **nested function call**.

```
print(max(3, -2, 1, 5, 0))
```

```
5
```

Figure 4.43 Using a nested function call

```
print()

print(1)
```

```
1
```

```
print(1, 2)
```

```
1  2
```

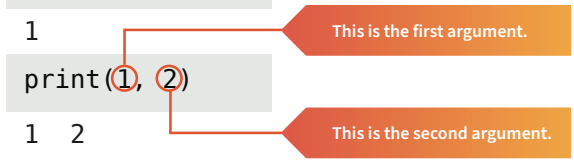


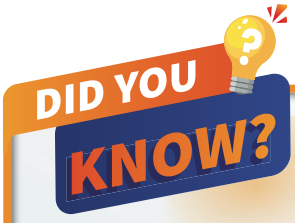
Figure 4.41 Using the `print()` function

KEY TERMS

Nested function call
Using the return value of one function call as an argument for another function call

Return value
The output of a function, or the value that a function call is treated as, after the instructions assigned to the function are completed

Source code can get difficult to read if there are too many nested function calls. For such cases, it is recommended to break the instruction into smaller pieces by assigning return values to intermediate variables before performing the outermost function call, as demonstrated earlier in Figure 4.42.



Besides identifying arguments based on position (first argument, second argument, etc.), arguments can also be identified by name. These are called "keyword arguments". For example, the `print()` function accepts multiple arguments and outputs the arguments separated using spaces by default:

```
print(1, 2, 3, 4)
```

```
1 2 3 4
```

Figure 4.44 The `print()` function separates arguments using spaces by default

However, you can specify a different separator using the `sep` keyword argument:

```
print(1, 2, 3, 4, sep=',')
```

```
1,2,3,4
```

Figure 4.45 The `print()` function accepts a `sep` keyword argument to specify a different separator

Some features of Python, such as using a different separator as shown above, can only be accessed using keyword arguments.

To learn about the arguments (including keyword arguments) that a function accepts, enter a question mark "?" before ar after the function name without any parentheses and press Shift-Enter. For example, to learn more about `print()`, you can enter "`?print`" into a code cell and press Shift-Enter.

4.6.2 Methods

A **method** is a function that is associated with a value. To use a method, type a value that is associated with the method, followed by a period (.), followed by the name of the method, and followed by zero or more comma-separated arguments enclosed within parentheses (see Figure 4.46). This is sometimes called "dot syntax". Just like in calling functions, arguments are assigned to new variables before running the method's instructions.

Values that have associated methods are called **objects**.

Syntax 4.4 Method Call

```
value.name_of_method()

value.name_of_method(argument_1)

value.name_of_method(argument_1,
argument_2)
```

Figure 4.46 Syntax for method calls

Method
A function that is associated with a value

Object
A value with associated data and/or methods

Python provides some additional functions through methods. For instance, in the following program, the `upper()` method associated with the `message` variable returns a copy of the `message` converted to uppercase:

```
message = "Hello, world!"
new_message = message.upper()
print(new_message)
```

```
HELLO, WORLD!
```

Figure 4.47 Using the `upper()` method

4.6.3 Operators

You have already seen operators used in many of our previous examples. For instance, the addition operator (+) takes two values, one to the left and one to the right, then returns the sum of the two values.

```
print(1 + 2)
```

```
3
```

Figure 4.48 Using the addition operator

Similarly, whenever we assign a value to a variable, we are actually using the assignment operator (=), which takes the value calculated on the right and stores it in the variable on the left:

```
year = 2024
```

Figure 4.49 Using the assignment operator

Most of the operators we will learn about take in exactly two values, one to the left and one to the right. These are called **binary operators** and the value(s) manipulated by an operator are called **operands.**

Some operators take in only one operand on the right, such as the negation operator (-) shown in Figure 4.50. These are called **unary operators**.

```
value = 65
```

```
print(-value)
```

```
-65
```

Figure 4.50 Using the negation operator

Operators and functions are similar in that they take in values, manipulate them, and usually return a value. While functions have names and use a standard calling syntax, operators usually look like symbols and the syntax for using them varies. However, because they are short and quick to type, operators are ideal for representing instructions that are frequently used.

KEY TERMS

Binary operator
An operator that takes in exactly two values

Operand
A value that is manipulated by an operator

Operator
A symbol for performing instructions like a function but using a different syntax

Unary operator
An operator that takes in exactly one value

QUICK CHECK 4.6

1. Identify whether each of the following code snippets shows a function call, a method call or the use of an operator:

a)
```
print(name, contact)
```

b)
```
len(description)
```

c)
```
numbers.append(3)
```

d)
```
cost / count
```

e)
```
f.readline()
```

f)
```
-result
```

2. Examine the following program:

```
print(abs(min(-4, 0, 1))
```

`abs()`, `min()` and `print()` are built-in Python functions.

State the order in which the `abs()`, `min()` and `print()` function calls are performed in the program.

(Hint: an inner function call must be completed first so its return value can be used as an argument for the outer function call.)

3. Examine the following program:

```
print(abs(min(-4, 0, 1) + 1))
```

`abs()`, `min()` and `print()` are built-in Python functions. The + operator is used to perform an addition operation.

State the order in which the `abs()`, `min()` and `print()` function calls and the addition operation are performed in the program.

4.7.7 Data Types

Recall that in a computer data is stored as bits and bytes. The same sequence of bytes can represent different types of data such as numbers or text, depending on how the bytes are interpreted. If data is interpreted wrongly, such as treating numbers as text or vice versa, it can lead to unintended behaviour.

To ensure that correct interpretations are used, Python associates every value with a data type. Some built-in data types in Python are shown in Table 4.12:

Data type	Python	Valid values	Example
Boolean	`bool`	True, False	`True`
Integer	`int`	Whole numbers	`1234`
Floating point	`float`	Real numbers	`12.34`
String	`str`	Text (can include digits)	`"ABC123"`
List	`list`	Multiple values	`[1,2,3]`
Dictionary	`dict`	Key-value pairs	`{12:'ABC'}`

Table 4.12 Built-in data types

Note that there are two different data types for numbers in Python, namely `int` and `float`, depending on whether it is necessary to represent non-whole numbers (i.e., real numbers). In addition, the data type for text in Python is called `str`.

These data types are discussed in further detail from sections 4.9 to 4.13.

4.8 Input and Output

LEARNING OUTCOMES

2.3.3 *Use the built-in functions: input() and print(), to perform interactive input/output using the keyboard and screen.*

4.8.1 Keyboard and Screen

As programmers, we can provide inputs (i.e., data for the program to process) directly in the source code as literals. However, computer users who are not programmers may not be able to edit a program's source code. For users to provide inputs when the program is run, we can instead use the `input()` function. This function prompts the user to enter text, which is then used as input for the program.

Figure 4.51 demonstrates asking for the user's name using `input()`, storing the user's response using a variable, then displaying the user's name on the screen using `print()`.

```
name = input('Enter your name: ')
print('Hello', name)
```

```
Enter your name: Bala
Hello Bala
```

Figure 4.51 Using the `input()` function

Note how `input()` outputs the given prompt "Enter your name: " then waits for the user to provide some text using the keyboard. The program will continue to wait until the user hits the enter key, then the entire `input()` function call is treated as the text that was entered just before pressing the enter key.

Table 4.13 summarises the `input()` and `print()` functions that are used to perform interactive input/output using the keyboard and screen.

Function	Argument(s)	Description	Examples
`input()`	An optional `str` to use as a prompt	Used to get input from the user with an optional prompt. Returns a `str` containing text that the user enters.	`s = input()` `print(s)` `Example` `Example` `s = input('Enter s:')` `print(s)` `Enter s: Example` `Example`

Function	Argument(s)	Description	Examples
`print()`	Any number of printable values	Used to display output. Displays arguments separated by spaces and starts a new line. Outputs an empty line if no arguments are given. Returns None.	`print('Example', 2024)` `print()` `print(2030)` `Example 2024` `2030`

Table 4.13 Keyboard and screen input and output functions

Note that `input()` returns what the user entered using the text data type `str`. If you need to perform calculations using the input, an additional step is needed to convert the `str` into a number data type (i.e., an `int` or `float`).

4.8.2 Files

The `input()` and `print()` functions have some disadvantages. For instance, the `input()` function typically requires the user to re-enter their responses each time it is called, which can be time-consuming. Similarly, the output of `print()` is saved in the notebook but can be inconvenient to extract for use in other programs, such as a spreadsheet. To address these disadvantages, an alternative approach is to use files on your computer for input and output.

We will cover how to use files for input and output when we discuss the `with` statement later in section 4.16.

QUICK CHECK 4.8

1. Write a program that uses `input()` and `print()` to solve the following problem:

Input	Output
• *name*: name of the user	• *name* displayed 3 times on 3 separate lines

Table 4.14 Input and output requirements for multiple-line name printing problem

2. Write a program that uses `input()` and `print()` to solve the following problem:

Input	Output
• *name*: name of the user	• *name* displayed 3 times on a single line, separated by spaces

Table 4.15 Input and output requirements for single-line name printing problem

4.9 Booleans

LEARNING OUTCOMES

2.3.6 *Use Boolean values with the operators: or, and, not.*

A Boolean (or `bool`) is a special data type used to represent logic-related data. Unlike other data types, there are only two valid `bool` values: `True` and `False`.

4.9.1 Boolean Literals

The literals for `bool` are also written as `True` and `False`:

```
seawater_is_salty = True
peanuts_are_nuts = False
print(seawater_is_salty)
print(peanuts_are_nuts)
```

```
True
False
```

Figure 4.52 Examples of valid `bool` literals

Note that both literals are case-sensitive and that lowercase `true` and `false` will produce errors instead:

```
seawater_is_salty = true
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 seawater_is_salty = true

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

```
peanuts_are_nuts = false
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[5], line 1
----> 1 peanuts_are_nuts = false

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

Figure 4.53 Examples of invalid `bool` literals

4.9.2 Boolean Operators

4.9.2.1 Equality

Python produces a `bool` value when we use the following equality operators to compare whether any two values are equal:

Operator	Name	Operand(s)	Description	Examples
==	Equality	Any two values	Returns `True` if the two values are equal and `False` otherwise.	`print(2024 == 2024)` `True` `print(2024 == '2024')` `False`
!=	Non-equality	Any two values	Returns `True` if the two values are not equal and `False` otherwise.	`print(2024 != 2024)` `True` `print(2024 != '2024')` `False`

Table 4.16 Equivalence operators

For Python to consider two values to be equal, they must be the same in both content and data type. For instance, note from the examples that although the whole number `2024` and text `'2024'` appear similar, Python does not consider them to be equal.

However, numbers are treated a little differently. If an `int` and a `float` are compared with each other, the `int` is converted to a `float` before the comparison is made. Furthermore, as you will learn in section 4.10.2, `floats` are not particularly accurate, so testing for equality with `floats` may give surprising results as seen in Figure 4.54. To avoid such issues, it is generally a good practice to use `ints` instead of `floats` (such as by storing prices in cents instead of dollars) if it is possible to do so.

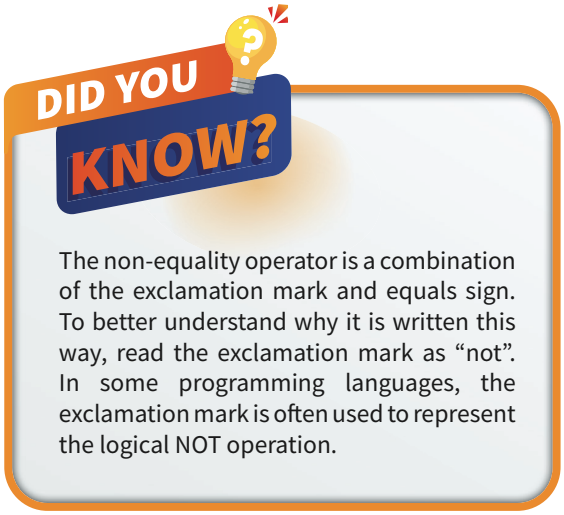
```
print(3 * 0.1 == 0.3)
```

```
False
```

```
x = 10000000000000000.0
print(x + 1 == x)
```

```
True
```

Figure 4.54 Using the equality operator with `floats` may give unexpected results



The non-equality operator is a combination of the exclamation mark and equals sign. To better understand why it is written this way, read the exclamation mark as "not". In some programming languages, the exclamation mark is often used to represent the logical NOT operation.

Note that the single equals sign is for assignment and double equals sign is for testing whether two values are equal. The two operators are not the same. For instance, the equals sign (=) does not have a return value while the equality operator (==) always has a `bool` return value of either `True` or `False`.

4.9.2.2 Logical

Boolean values can be manipulated using logical operators:

Operator	Name	Operand(s)	Description	Examples
`and`	Conjunction	Two `bool`s	Returns `True` if both values are true and `False` otherwise.	`print(True and True)` `True` `print(True and False)` `False`
`not`	Negation	A `bool`	Takes a `bool` and returns its opposite value.	`print(not True)` `False` `print(not False)` `True`
`or`	Disjunction	Two `bool`s	Returns `True` if either one or both values are true and `False` otherwise.	`print(True or True)` `True` `print(True or False)` `True`

Table 4.17 Logical operators

These logical operators allow multiple pieces of information to be combined to make a conclusion or decision. Suppose `is_raining` is a `bool` that represents whether or not it is currently raining, and `have_umbrella` is a `bool` that represents whether or not there is an umbrella available. The decision to leave home may then be written using logical operators.

```
# Program 4.3: logical_operators.ipynb

1 is_raining = True
2 have_umbrella = False
3 leave_home = (not is_raining) or (is_raining and have_umbrella)
4 print(leave_home)
```

```
False
```

Figure 4.55 Example of using logical operators

The statement on Line 3 translates very naturally into English as "we can leave home if it is not raining or if it is raining and we have an umbrella". Lines 1 and 2 specify the specific scenario where `is_raining` is `True` and `have_umbrella` is `False`. The output is thus `False` as leaving home in this scenario would probably get us drenched!

DID YOU KNOW?

We can use what we learned in Chapter 3 on the properties of AND and OR to simplify Line 3. Converting the Python code into a Boolean statement, we get:

$$\begin{aligned} leave\_home &= \overline{is\_raining} + (is\_raining \cdot have\_umbrella) \\ &= (\overline{is\_raining} + is\_raining) \cdot (\overline{is\_raining} + have\_umbrella) \\ &= 1 \cdot (\overline{is\_raining} + have\_umbrella) \\ &= \overline{is\_raining} + have\_umbrella \end{aligned}$$

Hence, Line 3 can be simplified to:

```
leave_home = (not is_raining) or have_umbrella
```

Figure 4.56 Simplified example of using logical operators

This can be understood as always being able to leave home if we have an umbrella (regardless of weather). Otherwise, we need check the weather to ensure that it is not raining. This is identical to the original statement.

QUICK CHECK 4.9

1. State whether each of the expressions below evaluate to `True` or `False`.
a) `not True` **b)** `not False` **c)** `1 == 1` **d)** `1 == 1.0` **e)** `1 == '1'` **f)** `0 != 0` **g)** `0 != 1`

2. At a water treatment plant, an alarm rings if the water level in a tank is full and the drainage pipe is closed, unless an override switch is turned on.

It is given that `tank_full` is a `bool` equal to whether the tank is full, `drainage_open` is a `bool` equal to whether the drainage pipe is open and `override_on` is a `bool` equal to whether the override switch is turned on.

Which of the following Boolean expressions is equal to whether the alarm is ringing?
a) `(tank_full and drainage_open) or not override_on`
b) `(tank_full and drainage_open) and override_on`
c) `(tank_full and not drainage_open) and not override_on`
d) `(tank_full and not drainage_open) or override_on`

3. A programmer wishes to obtain a Boolean that is `True` only if an `int` variable, `num`, is either `-1` or any value other than 10 or 20.

Fill in the blanks for the required Boolean expression below using "`and`" and "`or`" only.

```
num == -1 ___________ (num != 10 ___________ num != 20)
```

4.10 Integers and Floating-Point Numbers

LEARNING OUTCOMES

2.3.7 *Use integer and floating-point values with appropriate operators and built-in functions (limited to those mentioned in the Quick Reference Guide) to perform:*
- *Addition, subtraction, multiplication, division, modulo and exponentiation*
- *Rounding (normal, up, down, towards zero)*
- *Calculation of square roots*
- *Generation of ranges*
- *Generation of random integers / floats*
- *Conversion to and from strings*

2.3.5 *Use the import command to load and make additional variables and functions available for use.*

4.10.1 Integer Literals

Integers (or `int`) comprise of positive and negative whole numbers as well as zero. Its literal format is an optional sign (+ or -) followed by the number's digits written out.

Figure 4.57 shows some examples of valid `int` literals.

```
2024
0
-1965
```

Figure 4.57 Examples of valid `int` literals

Integers in Python have no limit, positive or negative, in the whole numbers they can represent.

4.10.2 Floating-Point Number Literals

A **floating-point number** (or `float`) is a real number.

There are two common literal formats for `float`:
1. An optional sign followed by the number's digits written out, with a decimal point included explicitly. (If no decimal point is included, Python will treat the literal as an `int` instead.) Example: `123.4`
2. The form AeB or AEB to represent A × 10B. Either A, B or both may start with an optional sign. Example: `1.234e2`

Figure 4.58 shows some examples of valid `float` literals:

```
2024.0
-19.65
6.02E23
-2.03e3
```

Figure 4.58 Examples of valid `float` literals

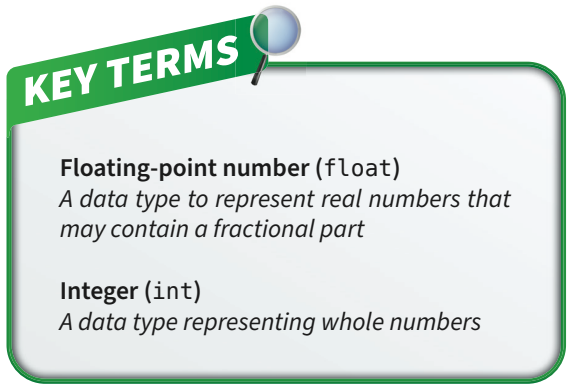
Due to the format in which they are stored, `floats` are not especially accurate, as shown in Figure 4.59:

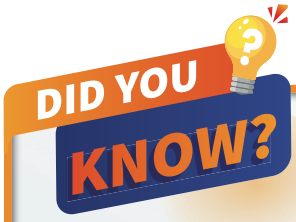
```
print(3 * 0.1)
```

```
0.30000000000000004
```

Figure 4.59 Example of an inaccurate floating-point calculation

This inaccuracy may cause surprising results when testing for equality, as mentioned previously in section 4.9.2.1.





The most positive and most negative limits that can be represented using a `float` are approximately 1.79×10^{308} and -1.79×10^{308} respectively. Beyond these limits, the `float` will get converted into the special value of infinity (represented by `inf` or `-inf`).

```
print(1.79e308)
```

```
1.79e+308
```

```
print(1.80e308)
```

```
inf
```

```
print(-1.79e308)
```

```
-1.79e+308
```

```
print(-1.80e308)
```

```
-inf
```

Figure 4.60 Demonstration of `float` limits

There is also a limit for how small a positive non-zero number can be. If a number gets too close to 0, it will get converted to 0. This is called "underflow".

```
print(1e-323)
```

```
1e-323
```

```
print(1e-324)
```

```
0.0
```

Figure 4.61 Demonstration of `float` limits

You do not need to memorise these limits. However, it is useful to know they exist in case you run into them.

4.10.3 Mathematical Operators

Mathematical operators are commonly used to operate on `int` and `float` data types.

4.10.3.1 Comparison

Python produces a `bool` value when we use any of the following comparison operators:

Operator	Name	Operand(s)	Description	
<	Less than	Two numbers	Returns `True` if the number on the left is less than the number on the right and `False` otherwise.	`print(2024 < 2030)` `True` `print(2024 < 2024)` `False` *Note: < also works with `strs` and `lists`. See Table 4.33 and Table 4.41.*
<=	Less than or equal to	Two numbers	Returns `True` if the number on the left is less than or equal to the number on the right and `False` otherwise.	`print(2024 <= 2030)` `True` `print(2024 <= 2024)` `False` *Note: <= also works with `strs` and `lists`. See Table 4.33 and Table 4.41.*
>	Greater than	Two numbers	Returns `True` if the number on the left is greater than the number on the right and `False` otherwise.	`print(2030 > 2024)` `True` `print(2024 > 2024)` `False` *Note: > also works with `strs` and `lists`. See Table 4.33 and Table 4.41.*
>=	Greater than or equal to	Two numbers	Returns `True` if the number on the left is greater than or equal to the number on the right and `False` otherwise.	`print(2030 >= 2024)` `True` `print(2024 >= 2024)` `True` *Note: >= also works with `strs` and `lists`. See Table 4.33 and Table 4.41.*

Table 4.18 Comparison operators for `ints` and `floats`

Note that `ints` can be compared to `floats` and vice versa using these operators as they both represent numbers. Otherwise, in general these comparison operators require both operands to have the same data type.

4.10.3.2 Arithmetic

You have already seen some of the operators used for performing calculations or arithmetic, as shown in Table 4.19:

Operator	Name	Operand(s)	Description	Examples
+	Addition	Two numbers	Returns the sum of two numbers.	`print(9 + 4)` `13` `print(9.0 + 4)` `13.0` *Note: + is also used as the concatenation operator for `strs` and `lists`. See Table 4.34 and Table 4.42.*
-	Negation	One number	Returns the negated number.	`print(-(9 + 4))` `-13` `print(-(9.0 + 4))` `-13.0`
-	Subtraction	Two numbers	Returns the difference of two numbers.	`print(9 - 4)` `5` `print(9.0 - 4)` `5.0`
*	Multiplication	Two numbers	Returns the product of two numbers.	`print(9 * 4)` `36` `print(9.0 * 4)` `36.0` *Note: * is also used as the repetition operator for `strs` and `lists`. See Table 4.34 and Table 4.42.*

Operator	Name	Operand(s)	Description	Examples
/	Division	Two numbers	Returns the number on the left divided by the number on the right.	`print(9 / 4)` `2.25` `print(9.0 / 4)` `2.25`
//	Floor division	Two numbers	Returns the number on the left divided by the number on the right, rounded down to the nearest integer.	`print(9 // 4)` `2` `print(9.0 // 4)` `2.0`
%	Modulus (remainder)	Two numbers	Returns the remainder when the number on the left is divided by the number on the right.	`print(9 % 4)` `1` `print(9.0 % 4)` `1.0`
**	Exponentiation (power)	Two numbers	Returns the number on the left raised to the power of the number on the right.	`print(9 ** 4)` `6561` `print(9.0 ** 4)` `6561.0`

Table 4.19 Arithmetic operators

As `float` values are not as accurate as `int` values, any mathematical operation that mixes the two types will force the calculation to use the one with lower accuracy and produce a `float` result. For instance, all the examples in Table 4.19 that involve `9.0` return `float` values because `9.0` is a `float` value.

The division operator (`/`) is an exception to this rule as the division of whole numbers often does not result in a whole number. In Python, when we divide an `int` with another `int` we will get back a `float`. If we want to perform a division that rounds down to the nearest integer, we would use floor division (`//`) instead.

While the result of floor division is always a whole number, if any of the values used in the calculation is a `float`, then the result will still be a `float`. In such cases, you can safely convert the result to an `int` using the `int`() function covered later in Table 4.21.

```
print(2024 / 1000)
```
```
2.024
```
```
print(2024 // 1000)
```
```
2
```

Figure 4.62 Using the floor division operator

```
print(2024 // 1000.0)
```
```
2.0
```
```
print(int(2024 // 1000))
```
```
2
```

Figure 4.63 The floor division operator may return a `float` value

The modulus operator (%) is particularly useful for testing whether a number is odd or even. If `x % 2` is 1, then x is odd. Otherwise, `x % 2` is `0` and x must be even.

We often need to use a variable for a calculation and then update the variable with the result. For convenience, Python provides additional or "augmented" assignment operators to help save some typing when performing this task.

In the following table, the code in the second and third columns are equivalent:

Operator	Example	Equivalent
`+=`	`x += a`	`x = x + a`
`-=`	`x -= a`	`x = x - a`
`*=`	`x *= a`	`x = x * a`
`/=`	`x /= a`	`x = x / a`
`//=`	`x //= a`	`x = x // a`
`%=`	`x %= a`	`x = x % a`
`**=`	`x **= a`	`x = x ** a`

Table 4.20 Assignment operators that work with `int` and `float` data types

4.10.4 Built-In Mathematical Functions

Python provides a wide variety of built-in functions to perform various mathematical operations. Table 4.21 summarises the most common mathematical functions in Python.

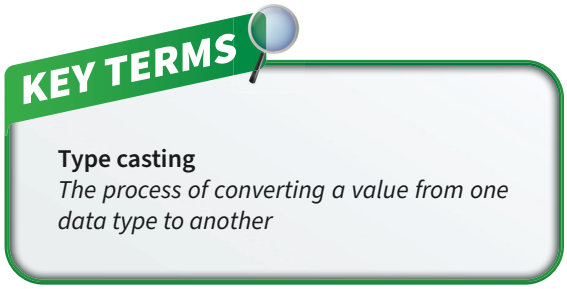
Operator	Name	Description	Examples
`abs()`	A number (`int` or `float`)	Returns an `int` or a `float` equal to the absolute value of the argument.	`print(abs(2024))` `2024` `print(abs(-2024))` `2024`
`int()`	Usually a `float` or `str`	Returns an `int` equal to the argument converted to an integer (if possible). For `float`s, the conversion cuts off the digits after the decimal point.	`print(int(2.024))` `2.0` `print(int('2024'))` `2024` `print(int(3.9))` `3` `print(int(-3.9))` `-3`

Operator	Name	Description	Examples
`float()`	Usually an `int` or a `str`	Returns a `float` equal to the argument converted to a floating-point number (if possible).	`print(float(2))` `2.0` `print(abs(-2024))` `2.024`
`max()`	Multiple numbers (`int` or `float`)	Returns the maximum value out of the given arguments.	`print(max(2, -2, 4))` `4` *Note: `max()` also works with `lists`. See Table 4.45.*
`min()`	Multiple numbers (`int` or `float`)	Returns the minimum value out of the given arguments.	`print(max(2, -2, 4))` `4` *Note: `min()` also works with `lists`. See Table 4.45.*
`pow()`	Two numbers (each of which can be `int` or `float`)	Returns a `float` equal to the first argument raised to the power of the second argument.	`print(pow(2024, 2))` `4096576` `print(pow(2, 2.024))` `4.067098693167825` *Note: The ** operator is equivalent to `pow()`. See Table 4.19.*
`range()`	One `int`	Returns a sequence of `ints` starting from 0 up to but not including the argument.	`print(list(range(3)))` `[0, 1, 2]` *Note: `list()` converts the range into a `list` so its values can be printed easily. See Table 4.45.*
`range()`	Two `ints`	Returns a sequence of `ints` starting from the first argument up to but not including the second argument.	`print(list(range(4, 8)))` `[4, 5, 6, 7]` *Note: `list()` converts the range into a `list` so its values can be printed easily. See Table 4.45.*
`range()`	Three `ints`	Returns a sequence of `ints` starting from the first argument up to but not including the second argument, in increments of the third argument.	`print(list(range(3, 20, 4)))` `[3, 7, 11, 15, 19]` *Note: `list()` converts the range into a `list` so its values can be printed easily. See Table 4.45.*

Table 4.21 Built-in mathematical functions

Operator	Name	Description	Examples
`round()`	A `float`	Returns an `int` equal to the argument rounded to the nearest integer.	`print(round(2.024))` `2` `print(round(-2.024))` `-2`
`round()`	A `float` and an `int`	Returns a `float` equal to the first argument rounded to the number of decimal places indicated by the second argument.	`print(round(2.024, 2))` `2.02` `print(round(-2.024, 2))` `-2.02`

Programmers often need to convert values of one type to another. The process of intentionally converting a value from one data type to another is called **type casting**. In Python, we do this using the name of the desired data type like a function. For instance, the `int()` function can convert `floats` into `ints`. However, note that the conversion is done by truncating all digits after the decimal point, as shown by the examples provided in Table 4.21. This is different from `round()`, which rounds a `float` to the nearest whole number.



We will discuss more details about `round()` and other functions that perform rounding later in section 4.10.6.

The `int()` and `float()` functions are also important for converting `strs` to numbers. Figure 4.64 shows that while `"123"` and `int("123")` both appear as 123 when printed, `"123"` is text and cannot be used to perform calculations while `int("123")` is the actual number 123 and can be used to perform calculations.

```
print("123")
```
```
123
```
```
print(int("123"))
```
```
123
```
```
print("123" + 1)
```
```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 print("123" + 1)

TypeError: can only concatenate str (not "int") to str
```
```
print(int("123") + 1)
```
```
124
```

Figure 4.64 Using the `int()` function to convert a `str` into an `int`

As mentioned in section 4.8.1, the `input()` function returns a `str`. If we want to the use the input as an `int` or `float` for calculations, we need to use either `int()` and `float()` accordingly. For instance, Figure 4.65 shows a program that accepts user input to calculate the average mass of a steel beam given the total mass and number of steel beams:

```
# Program 4.4: average_mass.ipynb

total_mass = float(input('Enter total mass (kg): '))
beams = int(input('Enter number of beams: '))
print('Average mass of beam (kg):', total_mass / beams)
```

```
Enter total mass (kg): 900.8
Enter number of beams: 40
Average mass of beam (kg):    22.52
```

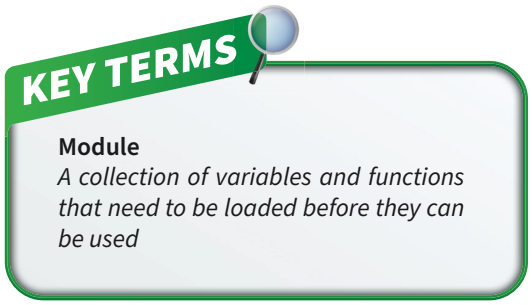
Figure 4.65 Converting user input into numbers using `int()` and `float()`

4.10.5 Using the math Module

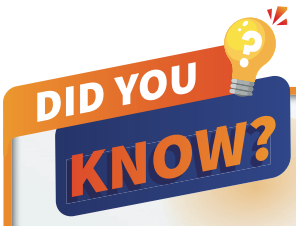
A **module** is a collection of variables and functions that needs to be loaded before the variables and functions can be used.

While convenient, keeping every variable and function loaded and ready for use has two major disadvantages:

1. Loading variables and functions and making them available takes up time and computer memory.
2. It becomes harder to choose names for variables and functions as names that were chosen may overwrite or clash with ones provided by Python or chosen by other programmers.



Not every program requires all the variables and functions that are available. Hence, Python provides some of its functions through modules.



There are many Python modules available for use, both included with Python and downloadable online as "packages". Some common Python packages that you may find online are shown in Table 4.22.

Package name	Description
`numpy`	For performing mathematical operations on tables of data
`pandas`	For analysing and manipulating data in various formats; built on top of `numpy`
`matplotlib`	For creating charts and graphs
`openpyxl`	For reading, writing and manipulating Microsoft Excel files

Table 4.22 Common Python packages

The module name for each of these packages is usually the same as the package name. In Jupyter Desktop, packages can be conveniently downloaded and installed using the Python package manager `pip`, as shown in Figure 4.66:

```
%pip install numpy
```

Figure 4.66 Installing the numpy package in Jupyter Desktop

Alternatively, we may create our own modules by simply assigning variables and defining functions we want to reuse in a `py` file in the same folder as our notebook; its filename (without the extension) will be used as the module name.

Some of the less commonly used mathematical functions are provided using a module named `math`. To load and use the `math` module, we use the `import` keyword like this:

```
import math
```

Figure 4.67 Importing the `math` module

To access and use the variables and/or functions within the module, we use the same "dot syntax" as accessing the methods of an object:

Syntax 4.5 Acessing Module Contents

```
name_of_module.name_of_variable()

name_of_module.name_of_function()

name_of_module.name_of_function(argument_1)

name_of_module.name_of_function(argument_1, argument_2)
```

Figure 4.68 Syntax for accessing module contents

For instance, the `math` module has a `sqrt()` function that calculates the square root of a number. Figure 4.69 shows how to use this function to calculate the square root of 9:

```
import math
result = math.sqrt(9)
print(result)
```

```
3.0
```

Figure 4.69 Accessing the `sqrt()` function of the `math` module

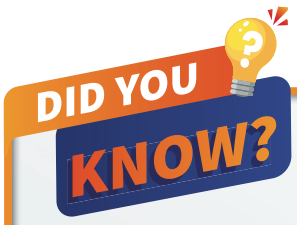
Sometimes, a module name may be too long and troublesome to be typed repeatedly each time we use it. Alternatively, the module name may already be used as another function or variable in our code.

In such cases, we can use a different name for the module when importing it by using the `import` and `as` keywords. For instance, we can import the math module using the name `my_math_module`, as shown in Figure 4.70:

```
import math as my_math_module
result = my_math_module.sqrt(9)
print(result)
```

```
3.0
```

Figure 4.70 Importing the `math` module using the name `my_math_module`



Python also comes with a module named `turtle` that is useful for learning programming, drawing graphics, and creating simple games. The module is inspired by the Logo programming language and is meant to simulate a robotic drawing toy called a "turtle", as shown in Figure 4.71. The turtle robot and Logo were both widely used for introducing children to programming.

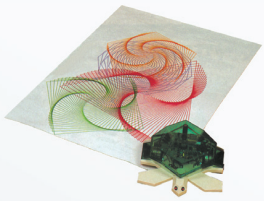


Figure 4.71 Example of a robot "turtle"

In the `turtle` module, there are commands for moving the turtle forward and turning the turtle left or right by a given number of degrees. For example, the following program instructs the turtle to draw a square in a separate window:

```
import turtle
turtle.forward(100)
turtle.right(90)
turtle.forward(100)
turtle.right(90)
turtle.forward(100)
turtle.right(90)
turtle.forward(100)
turtle.done()
```

Figure 4.72 Drawing a square using the `turtle` module

(Important: After running the cell, you must close the turtle window to continue using JupyterLab. If you encounter a `Terminator` error trying to run `turtle` commands after closing the window, just try running the cell again.)

We will use the `turtle` module to provide some optional examples later in this chapter.

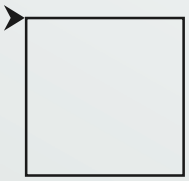


Figure 4.73 Square drawn using the `turtle` module

The syntax for `import` statements is summarised in Figure 4.74:

Syntax 4.6 import Statement

```
import name_of_module

import name_of_module as new_name
```

Figure 4.74 Syntax for `import` statement

Python modules may contain a large number of functions and variables. Sometimes, it is simpler to just import specific functions and variables that need to be used, rather than the entire module.

To do this, we can make use of the `from` and `import` keywords. For instance, Figure 4.75 shows how to import just the `sqrt()` function from the `math` module. Note that when using this form of the `import` statement, we access the function or variable name directly and it is not necessary to use the "dot syntax" described previously in Figure 4.68:

```
from math import sqrt
result = sqrt(9)
print(result)
```

```
3.0
```

Figure 4.75 Importing just the `sqrt()` function of the `math` module

Just like with the first form of the `import` statement, we can use a different name for the imported variable or function using the `as` keyword, as shown in Figure 4.76:

```
from math import sqrt as my_square_root
result = my_square_root(9)
print(result)
```

```
3.0
```

Figure 4.76 Importing just the `sqrt()` function of the `math` module using the name `my_square_root`

The syntax for this alternate form of `import` statements is summarised in Figure 4.77:

Syntax 4.7 import Statement (Alternate Form)

```
from name_of_module import function_name
from name_of_module import function_name as new_name
```

Figure 4.77 Syntax for alternate form of `import` statement

After importing the `math` module, the following functions will become available:

Function	Argument(s)	Description	Examples
`math.ceil()`	Usually a `float`	Returns an `int` equal to the smallest integer greater than or equal to the argument.	`print(math.ceil(2.024))` 3 `print(math.ceil(-2.024))` -2
`math.floor()`	Usually a `float`	Returns an `int` equal to the largest integer smaller than or equal to the argument.	`print(math.floor(2.024))` 2 `print(math.floor(-2.024))` -3
`math.sqrt()`	A number (`int` or `float`)	Returns a `float` equal to the square root of the argument.	`print(math.sqrt(2024))` 44.98888751680797
`math.trunc()`	Usually a `float`	Returns an `int` equal to the argument with digits after the decimal point cut off (i.e., truncated).	`print(math.trunc(2.024))` 2 `print(math.trunc(-2.024))` -2 *Note: Using `int()` with a `float` is equivalent to using `math.trunc()`. See Table 4.21.*

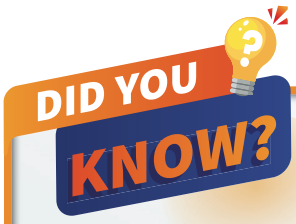
Table 4.23 Functions in the `math` module

Using any of these functions without importing the `math` module will result in an error. In the example below, Python does not understand what the name "`math`" means as it does not correspond to an existing variable or loaded module:

```
print(math.ceil(20.24))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 print(math.ceil(20.24))

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

Figure 4.78 Using the `math` module before it is imported will result in an error



We can use the built-in `dir()` function to list all the function and variable names in a module. For instance, Figure 4.79 shows how to list everything in the `math` module:

```
import math
print(dir(math))
```

Figure 4.79 Listing all the function and variable names in the `math` module

In this textbook, we will focus on using modules to access functions and not variables. However, from the list you should see that some important mathematical values such e and π are available in the `math` module as `math.e` and `math.pi` respectively.

4.10.6 Rounding Functions

A number of functions such as `math.ceil()`, `math.floor()`, `round()` and `math.trunc()` are similar in that they can convert `floats` to `ints`. This conversion is performed by rounding `floats` such that the resulting `ints` are close in value to the original `floats`. However, each function uses a different rounding method. Understanding the difference between these functions is important, because using the correct function can greatly reduce the number of steps needed for a solution.

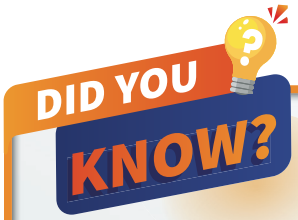
For `round()`, the argument is always rounded to the nearest whole number. However, an argument that is exactly halfway between two integers (such as 0.5) may be rounded either up or down. Table 4.24 demonstrates how `round()` can appear to give unpredictable results – notice that 0.5 is rounded down while 1.5 is rounded up.

x	round(x)	Explanation
−1.5	−2	Unpredictable since −1.5 is halfway between −2 and −1
−1.0	−1	Exact
−0.75	−1	Rounded down since −0.75 is nearer to −1 than 0
−0.5	0	Unpredictable since −0.5 is halfway between −1 and 0
−0.25	0	Rounded up since −0.25 is nearer to 0 than −1
0.0	0	Exact
0.25	0	Rounded down since 0.25 is nearer to 0 than 1
0.5	0	Unpredictable since 0.5 is halfway between 0 and 1
0.75	1	Rounded up since 0.75 is nearer to 1 than 0
1.0	1	Exact
1.5	2	Unpredictable since 1.5 is halfway between 1 and 2

Table 4.24 Behaviour of the round( ) function for numbers around zero

```
print(round(-1.0))
-1
print(round(0.5))
0
print(round(0.75))
1
print(round(1.5))
2
```

Figure 4.80 Using the round( ) function



The round( ) function only appears to give unpredictable results. In reality, round( ) will always return the even integer if its argument is exactly halfway between two integers. For example, round(0.5) will return 0 and round(1.5) will return 2.

On the other hand, `math.ceil()` always rounds up. Take note that this means a negative non-integer argument will end up becoming less negative.

x	math.ceil(x)	Explanation
−1.0	−1	Exact
−0.75	0	Rounded up
−0.5	0	Rounded up
−0.25	0	Rounded up
0.0	0	Exact
0.25	1	Rounded up
0.5	1	Rounded up
0.75	1	Rounded up
1.0	1	Exact

Table 4.25 Behaviour of the `math.ceil()` function for numbers around zero

```
import math
print(math.ceil(-1.0))
```

```
-1
```

```
print(math.ceil(0.5))
```

```
1
```

```
print(math.ceil(0.75))
```

```
1
```

Figure 4.81 Using the `math.ceil()` function

Conversely, `math.floor()` always rounds down. Take note that this means a negative non-integer argument will end up becoming more negative.

x	math.floor(x)	Explanation
-1.0	-1	Exact
-0.75	-1	Rounded down
-0.5	-1	Rounded down
-0.25	-1	Rounded down
0.0	0	Exact
0.25	0	Rounded down
0.5	0	Rounded down
0.75	0	Rounded down
1.0	1	Exact

Table 4.26 Behaviour of the `math.floor()` function for numbers around zero

```
import math
print(math.floor(-1.0))
```

```
-1
```

```
print(math.floor(0.5))
```

```
0
```

```
print(math.floor(0.75))
```

```
0
```

Figure 4.82 Using the `math.floor()` function

Finally, `math.trunc()` drops the digits of the argument after the decimal point (i.e., truncating the argument). This is equivalent to rounding towards zero.

x	math.trunc(x)	Explanation
-1.0	-1	Exact
-0.75	0	Rounded up towards 0
-0.5	0	Rounded up towards 0
-0.25	0	Rounded up towards 0
0.0	0	Exact
0.25	0	Rounded down towards 0
0.5	0	Rounded down towards 0
0.75	0	Rounded down towards 0
1.0	1	Exact

Table 4.27 Behaviour of the `math.trunc()` function for numbers around zero

```
import math
print(math.trunc(-1.0))
```
```
-1
```
```
print(math.trunc(-0.75))
```
```
0
```
```
print(math.trunc(0.75))
```
```
0
```
```
print(math.trunc(1.0))
```
```
1
```

Figure 4.83 Using the `math.trunc()` function

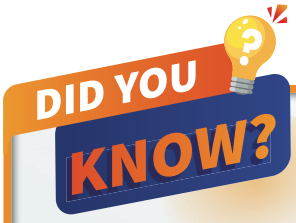
4.10.7 Using the random Module

Many computer games often require the computer to make a random choice or to generate a random number. To do this, Python provides a module named `random`:

Function	Argument(s)	Description	Examples
`random.randint()`	Two `int`s (the first argument should be less than or equal to the second argument)	Returns a random `int` between the two arguments (inclusive).	`print(random.randint(2, 7))` 6 `print(random.randint(2, 7))` 3 `print(random.randint(2, 7))` 2 *Note: The output of the above examples is random.*

Function	Argument(s)	Description	Example
`random.random()`	None	Returns a random `float` between 0 (inclusive) and 1 (exclusive).	`print(random.random())` `0.20814215199300812` `print(random.random())` `0.06804711169732913` `print(random.random())` `0.5070243505930644` *Note: The output of the above examples is random.*

Table 4.28 Functions in the `random` module



Run the following program that uses the `turtle` module to draw an artwork made up of four randomly-sized squares:

```
import random
import turtle

turtle.color('blue')
turtle.begin_fill()
length = random.randint(4, 10) * 10
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.end_fill()

turtle.color('green')
turtle.begin_fill()
length = random.randint(4, 10) * 10
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.end_fill()

turtle.color('red')
turtle.begin_fill()
length = random.randint(4, 10) * 10
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.end_fill()

turtle.color('orange')
turtle.begin_fill()
length = random.randint(4, 10) * 10
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.right(90)
turtle.forward(length)
turtle.end_fill()

turtle.done()
```

Figure 4.84 Drawing randomised artwork using the `turtle` module

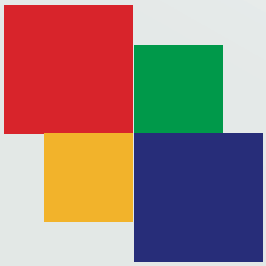


Figure 4.85 Randomised artwork drawn using the `turtle` module

Notice that the code is rather repetitive. We will see how to shorten it later.

QUICK CHECK 4.10

1. State whether each of the expressions below evaluate to `True` or `False`.
a) `0 > 1` **b)** `1 > 0` **c)** `2 > 0 and 2 < 1` **d)** `2 > 0 or 2 < 1`

2. What is the output of the following code?

```
product = 10.49 * -2
converted_to_int = int(product)
print(converted_to_int)
```

a) *-21* **b)** *20* **c)** *20* **d)** *21*

3. Predict the output of the following code:

```
remainder = 17 % 6
print(remainder)
```

4. Predict the output of the following code:

```
x = 1
x += 2
x *= 2
x -= 2
print(x)
```

5. For each value of x below, predict the result of the respective rounding functions. Check your answers by printing the actual return values in Python.

x	-20.24	-19.65	-0.1	-0.0	0.1	19.65	20.24
`math.ceil(x)`							
`math.floor(x)`							
`round(x)`							
`math.trunc(x)`							

Table 4.29 Results of various rounding functions

6. When two six-sided dice are rolled, the resulting sum can range from 2 to 12 (inclusive) with 7 being the most likely.

Write a program to simulate rolling two six-sided dice and output the resulting sum.

7. The area of a triangle with base *b* and height *h* is given by the formula $A = {}^1/_2bh$.

Write a program to solve the following triangle area calculation problem:

Input	Output
• *b*: base of triangle • *h*: height of triangle	• Area of triangle obtained by the formula $A = {}^1/_2bh$

Table 4.30 Input and output requirements for triangle area calculation problem

(*b* and *h* should be provided as inputs by the user and converted to `floats`.)

QUICK CHECK 4.10

7. The hypotenuse c of a right-angled triangle with other two sides of lengths a and b is given by Pythagoras's formula $c = \sqrt{a^2+b^2}$.

Write a program to solve the following hypotenuse calculation problem:

Input	Output
• a: length of triangle's side • b: length of triangle's side	• Length of triangle's hypotenuse obtained by Pythagoras's formula $c = \sqrt{a^2+b^2}$, rounded to 2 decimal places

Table 4.31 Input and output requirements for hypotenuse calculation problem

(a and b should be provided as inputs by the user and converted to `floats`.)

4.11 Strings

LEARNING OUTCOMES

2.3.8 *Use string values with appropriate operators, built-in functions and methods (limited to those mentioned in the Quick Reference Guide) to perform:*

- *Concatenation and repetition*
- *Extraction of characters and substrings (i.e., indexing and slicing)*
- *Conversion to upper and/or lower case*
- *Conversion of single characters to and from ASCII*
- *Testing of whether characters are letters only, lower-case letters only, upper-case letters only, digits only, spaces only and/or alphanumeric*
- *Testing of whether the string contains a substring*
- *Testing of whether the string starts with and/or ends with a substring*
- *Searching for the location of a substring*
- *Splitting of string into list of substrings based on either whitespace or a given delimiter*
- *Calculation of length*
- *Output formatting*

A string (or `str`) is used to represent text in Python. The name "string" is shorthand for the phrase "string of characters/letters".

KEY TERMS

String (`str`)
A data type to represent text as a sequence of characters or symbols

4.11.1 String Literals

4.11.1.1 Plain Strings

The most common form of a `str` literal consists of text enclosed by matching single-quotes (') or double-quotes (").

```
print('Single-quotes can be used')
```
```
Single-quotes can be used
```
```
print("Double-quotes can be used")
```
```
Double-quotes can be used
```
```
print("But the quotes must match!')
```
```
  Cell In[7], line 1
    print("But the quotes must match!')
          ^
SyntaxError: unterminated string literal (detected at line 1)
```

Figure 4.86 Examples of valid and invalid string literals

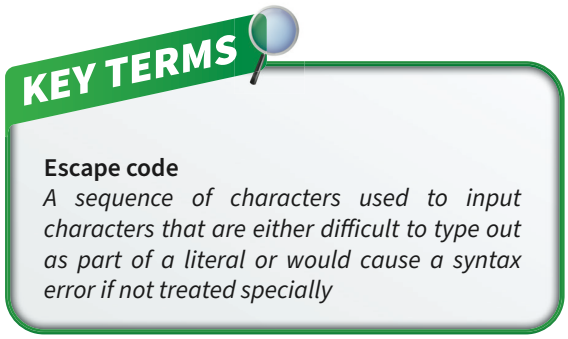
4.11.1.2 Escape Codes

Some `str`s may contain characters that are either difficult to type out as part of a literal or would cause a syntax error if not treated specially. To include such characters, we need to input them using the backslash (\) key and special **escape codes**.

Some common escape codes for string literals are shown below.

Excape code	Meaning
\\	Backslash (\)
\'	Single-quote (‘)
\"	Double-quote (“)
\n	Newline character
\t	Tab character
\	Ignore end of line

Table 4.32 Escape codes



The last escape code "\" represents a backslash that is the last character on a line of source code. This trailing backslash means to ignore the line break and treat the following line as if it continues on the current line.

Figure 4.87 shows some examples of how escape codes can be used:

The \n escape code breaks the string up into two lines.

```
print('Line 1\nLine 2')
```

```
Line 1
Line 2
```

Python ignores the line break as the first line ends with backslash

```
print('abc\
def')
```

```
abcdef
```

The single quote in "it's" is escaped and thus does not end the string prematurely

```
print('It\'s a small world.')
```

```
It's a small world.
```

Figure 4.87 Examples of how escape codes are used

4.11.1.3 Triple-Quoted Strings

For string literals that contain a large number of newlines, Python supports "triple-quoted strings" where the text is surrounded by matching groups of *three* single-quotes or double-quotes. These string literals can have multiple lines and the newlines are preserved, as shown in Figure 4.88:

```
# Normal string literals cannot include newlines without an escape code.
s = 'Line 1
Line 2'
```

```
  Cell In[2], line 2
    s = 'Line 1
        ^
SyntaxError: unterminated string literal (detected at line 2)
```

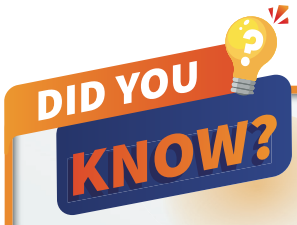
```
# However, triple-quoted strings can!
s = '''Line 1
Line 2'''
print(s)
```

```
Line 1
Line 2
```

```
print("""Either single-quotes are double-quotes can be used.
The literal ends when exactly three matching quotation marks are found,
so it's okay to use "lone" quotation marks without escaping them.
Normal escape codes\ncan also be used.""")
```

```
Either single-quotes are double-quotes can be used.
The literal ends when exactly three matching quotation marks are found,
so it's okay to use "lone" quotation marks without escaping them.
Normal escape codes
can also be used.
```

Figure 4.88 Examples of how triple-quoted strings are used



For string literals that contain a large number of backslashes, Python supports "raw strings" that are prefixed by "r" or "R". These string literals generally leave escape codes unchanged and treat the backslash as a normal character, as shown in Figure 4.89.

```
print(r"Escape \n codes \' do \" nothing \\ here!")
```

```
Escape \n codes \' do \" nothing \\ here!
```

Figure 4.89 Example of raw string

This format is especially useful for specifying file or folder locations on Windows where backslashes are used to separate folder names (e.g., `C:\Users\Example\Desktop`).

4.11.1.4 Formatted String Literals (f-strings)

Finally, we often want to create strings that seamlessly include other values. Python provides an easy way to do this using "formatted string literals" that are prefixed by "f" or "F". They are also called "f-strings". Each set of curly braces {} in the literal must contain a Python expression and is called a **replacement field**, or just field for short. When Python creates a string from the literal, all the expressions are evaluated, and each field is replaced by the value of the expression it contains. This can be used to easily create strings that include the value of variables or simple calculations, as shown in Figure 4.90:

```
year = 2024
print(f'The year is {year}')
print(f"The next year is {year + 1}")
```

```
The year is 2024
The next year is 2025
```

Figure 4.90 Example of formatted string literal

To "escape" the curly braces so that they are not treated as a field, use double braces so that it looks like "{{" or "}}".

```
year = 2024
print(f'Using {{year}} in a f-string produces {year}')
```

```
Using {year} in a f-string produces 2024
```

Figure 4.91 Escaping curly braces in an f-string

The syntax for replacement fields in formatted string literals is summarised in Figure 4.92:

Syntax 4.8 Replacement field (formatted string literals)

```
{expression}
```

Figure 4.92 Syntax for replacement fields (formatted string literals)

Values can also be included in strings using type casting and string concatenation (described in section 4.11.2) or the `str.format()` method (described in section 4.11.5).

KEY TERMS

Replacement field
A placeholder denoted by a set of curly braces that will be replaced with a value

4.11.2 String Operators

4.11.2.1 Comparison

The comparison operators in Table 4.18 for `ints` and `floats` also work with `strs`, as shown in Table 4.33. When comparing strings, the results are determined by comparing the ASCII codes of the first pair of differing characters (i.e., ASCII order). If no difference is found, the comparison is based on the lengths of the strings.

Operator	Name	Operand(s)	Description	Examples
<	Less than	Two `strs`	Returns `True` if the `str` on the left is less than the `str` on the right and `False` otherwise.	`print('A' < 'Z')` `True` `print('a' < 'Z')` `False` `print('A' < 'App')` `True` `print('App' < 'App')` `False` *Note: < also works with numbers and `lists`. See Table 4.18 and Table 4.41.*
<=	Less than or equal to	Two `strs`	Returns `True` if the `str` on the left is less than or equal to the `str` on the right and `False` otherwise.	`print('A' <= 'Z')` `True` `print('a' <= 'Z')` `False` `print('A' <= 'App')` `True` `print('App' <= 'App')` `True` *Note: <= also works with numbers and `lists`. See Table 4.18 and Table 4.41.*

Table 4.33 Comparison operators for `strs`

Operator	Name	Operand(s)	Description	Example
>	Greater than	Two strs	Returns True if the str on the left is greater than the str on the right and False otherwise.	`print('A' > 'Z')` `False` `print('a' > 'Z')` `True` `print('A' > 'App')` `False` `print('App' > 'App')` `False` *Note: > also works with numbers and lists. See Table 4.18 and Table 4.41.*
>=	Greater than or equal to	Two strs	Returns True if the str on the left is greater than or equal to the str on the right and False otherwise.	`print('A' >= 'Z')` `False` `print('a' >= 'Z')` `True` `print('A' >= 'App')` `False` `print('App' >= 'App')` `True` *Note: <= also works with numbers and lists. See Table 4.18 and Table 4.41.*

Table 4.33 Comparison operators for strs

In programming, we often want to arrange strings in alphabetical order. However, alphabetical order does not always correspond to ASCII order. For instance, in ASCII all upper-case letters have lower ASCII codes than lower-case letters and thus upper-case 'Z' will be "less than" lower-case 'a' using the operators in Table 4.33.

To achieve alphabetical ordering, we will need to process the strings beforehand to remove all non-letters and to convert all letters to either upper or lower-case.

4.11.2.2 Concatenation and Repetition

Sometimes, we may want to join two `str`s together or repeat a `str` multiple times. To perform these tasks, Python reuses the + and * arithmetic operators:

Operator	Name	Operand(s)	Description	Examples
+	Concatenation	Two `str`s	Returns a new `str` where the two `str`s are joined together.	`print("Comp" + "2024")` `Comp2024` *Note: + is also used for addition. See Table 4.19.* *Note: + also works with `lists`. See Table 4.42.*
*	Repetition	A `str` and an `int`, in any order	Returns a new `str` where the given `str` is repeated the given number of times.	`print("E.T." * 3)` `E.T.E.T.E.T.` `print(3 * "2024")` `202420242024` *Note: * is also used for multiplication. See Table 4.19.* *Note: * also works with `lists`. See Table 4.42.*

Table 4.34 Concatenation and repetition operators for `str`s

However, note that the + and * operators are exceptions and that `str`s do not work with arithmetic operators in general. For instance, trying to subtract one `str` from another `str` will result in an error.

```
"computer"-"er"
```

```
  Cell In[3], line 1
    "computer"-"er"
               ^
SyntaxError: invalid character in identifier
```

Figure 4.93 Not all arithmetic operators work with `str`

Like f-strings, we can use the + operator to produce strings that seamlessly include other values, including non-strings such as `int`s and `float`s. However, one complication is that to perform concatenation both operands must be `str`s or there will be an error:

```
"The year is " + 2024
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[1], line 1
----> 1 "The year is " + 2024

TypeError: can only concatenate str (not "int") to str
```

Figure 4.94 Concatenation requires both operands to be `str`s

The solution is to type cast the non-strings into `str`s, then use the + operator to concatenate the required `str`s together, as shown in Figure 4.95:

```
print("The year is " + str(2024))
```

```
The year is 2024
```

Figure 4.95 Using concatenation to include other values in a `str`

4.11.2.3 Indexing and Slicing

To perform more complex operations such as extracting characters from `str`s, we need to use the indexing and slicing operators.

We use the indexing operator in the form `str_value[i]`, where `i` is an `int` value. This `int` value is called the index. Starting with position 0 for the first character, position 1 for the second character, and so on, the indexing operator returns the character with the same position as the index.

For example, in Figure 4.96 and Figure 4.97, using the indexing operator with an index of 5 on the `str` `subject_name` returns the character 't', which has position 5 in the `str`.

```
subject_name = "Computing"
print(subject_name[5])
```

```
t
```

Figure 4.96 Using the indexing operator

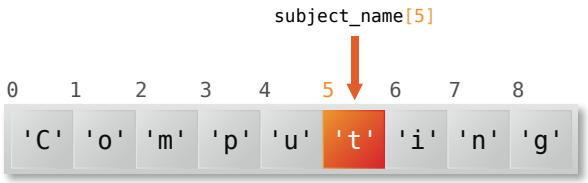


Figure 4.97 Using the indexing operator

If we want to extract a subset of characters from a `str` (instead of just a single character), we can use the slicing operator. We use this operator in the form `str_value[a:b]`, where `a` and `b` are `int` values indicating the start and stop indices respectively. This extracts the sequence of values or characters positioned from the start index up to but not including the stop index.

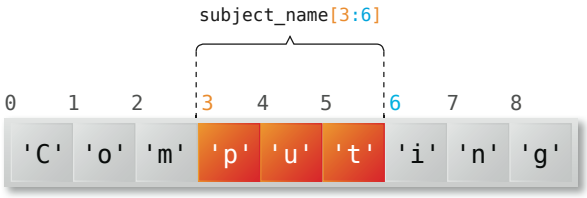


Figure 4.98 Using the slicing operator

If the start index is left out, it is treated as 0. If the stop index is left out, it is treated as the length of the `str`. Leaving out the stop index usually has the same meaning as making sure all remaining characters of the `str` are included in the return value.

Although not often used, the slicing operator can also accept an optional third `int` value in the form `sequence_name[a:b:c]`, where `a` and `b` are the start and stop indices while `c` is called the step. This extracts the sequence of values or characters positioned from the start index up to but not including the stop index in increments of the step.

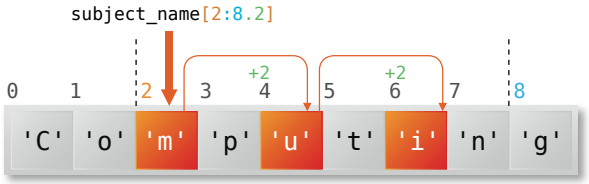


Figure 4.99 Using the slicing operator with a step

As before, if the start index is left out, it is treated as 0. If the stop index is left out, it is treated as the length of the `str` or `list`. If the step is left out, it is treated as 1.

```
print("Computing"[:8:2])
```

```
Cmui
```

```
print("Computing"[2::2])
```

```
muig
```

```
print("Computing"[2:8:])
```

```
mputin
```

Figure 4.100 Using the slicing operator with the start index, stop index or step left out

The indexing and slicing operators are summarised in Table 4.35:

Operator	Name	Operand(s)	Description	Example
`[i]`	Indexing	A `str` and an `int`	Returns the character in the `i`-th position of the `str`.	`s = "Computing"` `print(s[0])` `C` `print(s[3])` `p` `print(s[-1])` `g` *Note: The indexing operator also works with `lists` and `dicts`. See Table 4.43 and Table 4.46.*
`[a:b]`	Slicing	A `str` and two optional `ints`	Returns a `str` starting from the character at index a up to but not including the character at index b; a is treated as 0 if omitted and b is treated as the length of the sequence if omitted.	`s = "Computing"` `print(s[3:6])` `put` `print(s[:7])` `Computi` `print(s[3:])` `puting` *Note: The slicing operator also works with `lists`. See Table 4.43.*
`[a:b:c]`	Slicing with step	A `str` and three optional `ints`	Returns a `str` starting from the character at index a up to but not including the value at index b, in increments of c; a is treated as 0 if omitted, b is treated as the length of the sequence if omitted and c is treated as 1 if omitted.	`s = "Computing"` `print(s[2:8:2])` `mui` `print(s[:8:2])` `Cmui` `print(s[2::2])` `muig` `print(s[2:8:])` `mputin` *Note: The slicing with step operator also works with `lists`. See Table 4.43.*

Table 4.35 Indexing and slicing operators for `strs`

Using the indexing operator on the left-hand side of an assignment and trying to change the characters in a `str` will result in an error. This is because (like `int` and `float`) `str` is an immutable type, meaning that once a `str` is created, its contents cannot be changed. For instance, suppose we wish to change the first character of "Computing" to a K so that the string becomes "Komputing". Trying to do so directly results in an error:

```
subject_name = "Computing"
subject_name[0] = 'K'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[2], line 2
      1 subject_name = "Computing"
----> 2 subject_name[0] = 'K'

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

Figure 4.101 Failing to change a character in a `str`

The solution is to create a new `str` by concatenating pieces from the original `str`:

```
subject_name = "Computing"
subject_name = 'K' + subject_name[1:]
print(subject_name)
```

```
Komputing
```

Figure 4.102 Creating a new `str` to change the first letter of `subject_name`

The second line in Figure 4.102 creates a new `str` value of "Komputing" and assigns it back to `subject_name`. The previous `str` value of "Computing" is no longer referred to. Note that a new `str` has been created. This is not the same as changing characters in an existing `str`, which is not allowed by Python.

Another common task in solving problems is accessing the last value in a `str`. If the index for an indexing operator is negative, Python treats it as a position counting from the opposite end of the sequence, treating the last character as position −1, the second-to-last character as position −2, and so on:

```
subject_name = "Computing"
letter = subject_name[-1]
print(letter)
```

```
g
```

Figure 4.103 Accessing the last value by using a negative index

Negative indices also work with the slicing operator:

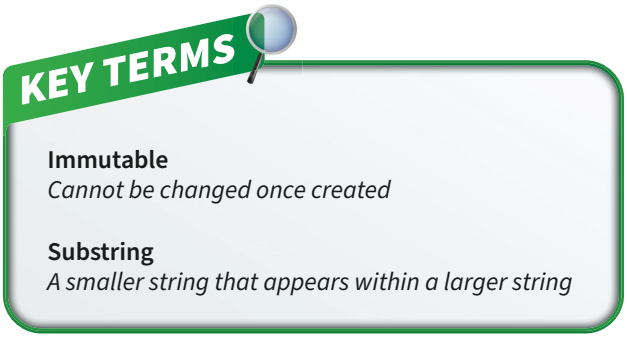
```
subject_name = "Computing"
letters = subject_name[4:-2]
print(letters)
```

```
uti
```

Figure 4.104 Using a negative index with the slicing operator

4.11.2.4 Membership

Finally, Python also provides a membership operator to test if a `str` appears in another `str`. Unlike the other operators you have learnt so far, the membership operator looks like a word (`in`) instead of a symbol, but it otherwise behaves the same way as other binary operators. The membership operator returns a Boolean `True` result if the `str` on the left appears in the `str` on the right as a **substring**. Otherwise the result is `False`.



Operator	Name	Operand(s)	Description	Example
`in`	Membership	Two `str`s	Returns `True` if the left `str` is a substring of the right `str` and `False` otherwise.	`print('4' in 'AB12')` `True` `print('4' in 'AB12')` `False` *Note: `in` also works with `lists` and `dicts`. See Table 4.44 and Table 4.47.*
`not in`	Non-Membership	Two `str`s	Returns `True` if the left `str` is not a substring of the right `str` and `False` otherwise.	`print('2' not in 'AB12')` `False` `print('4' not in 'AB12')` `True` *Note: `not in` also works with `lists` and `dicts`. See Table 4.44 and Table 4.47.*

Table 4.36 Membership operators for `str`s

4.11.3 String Functions

Table 4.37 summarises some common string functions in Python:

Function	Argument(s)	Description	Example
`chr()`	An `int`	Returns a `str` of length 1 containing the character with given ASCII code.	`print(chr(65))` `A`
`len()`	A `str`	Returns an `int` equal to the number of characters in the given `str`.	`print(len('CS'))` `2` *Note: `len()` also works with `lists` and `dicts`. See Table 4.45 and Table 4.48.*
`ord()`	A `str` (must be of length 1)	Returns an `int` equal to the ASCII code of the given character.	`print(ord('A'))` `65`
`str()`	Any value that can be converted to a `str`	Returns a `str` representation of the given argument.	`print('CS' + str(24))` `CS24`

Table 4.37 Built-in string functions

4.11.4 String Methods

Table 4.38 shows some common methods for `str` objects

String Method	Argument(s)	Description	Example
`endswith()`	A `str`	Returns a `bool` that indicates whether the `str` ends with the given argument.	`print('CS2024' .endswith('CS'))` `False` `print('CS2024' .endswith('24'))` `True`
`find()`	A `str`	Returns an `int` index where the given argument first appears in the `str`; -1 if not found.	`print('banana'.find('na'))` `2` `print('banana'.find('cs'))` `-1`

String Method	Argument(s)	Description	Example
`find()`	A `str` and an `int`	Returns an `int` index where the first argument first appears in the `str` starting from the second argument index; -1 if not found.	`print('banana' .find('na', 1))` `2` `print('banana' .find('na', 3))` `4` `print('banana' .find('na', 5))` `-1`
`find()`	A `str`, an `int` and an `int`	Returns an `int` index where the first argument first appears in the `str` starting from the second argument index up to but not including the third argument index; -1 if not found.	`print('SCSC' .find('CS', 0, 3))` `1` `print('SCSC' .find('CS', 1, 3))` `1` `print('SCSC' .find('CS', 0, 2))` `-1`
`format()`	Any number of values	Returns a new `str` with each field (such as '`{}`' or '`{0}`') in the `str` replaced by a corresponding argument.	`print('{} 2024' .format('CS'))` `'CS 2024'` `print('{0} {1}' .format(6, 5))` `'6 5'` `print('{1} {0}' .format(6, 5))` `'5 6'` *(see section 5.3.3 on formatting strings)*
`isalnum()`	None	Returns a `bool` that indicates whether every character in the `str` is alphanumeric (either a letter or a digit).	`print('CS2024'.isalnum())` `True` `print('CS 2024'.isalnum())` `False` `print('2024'.isalnum())` `True`
`isalpha()`	None	Returns a `bool` that indicates whether every character in the `str` is a letter.	`print('Computing' .isalpha())` `True` `print('Com pu ting' .isalpha())` `False`

String Method	Argument(s)	Description	Example
`isdigit()`	None	Returns a `bool` that indicates whether every character in the `str` is a digit.	`print('Computing' .isdigit())` `False` `print('2024'.isdigit())` `True`
`islower()`	None	Returns a `bool` that indicates whether every letter in the `str` is lowercase and there is at least one letter.	`print('computing' .islower())` `True` `print('com pu ting' .islower())` `True` `print('Computing' .islower())` `False` `print('2024'.islower())` `False`
`isspace()`	None	Returns a `bool` that indicates whether every character in the `str` is a space and there is at least one character.	`print('  '.isspace())` `True` `print(''.isspace())` `False`
`isupper()`	None	Returns a `bool` that indicates whether every letter in the `str` is uppercase and there is at least one letter.	`print('COMPUTING' .isupper())` `True` `print('COM PU TING' .isupper())` `True` `print('Computing' .isupper())` `False` `print('2024' .isupper())` `False`
`lower()`	None	Returns a `str` containing every character of the original `str` converted to lowercase.	`print('Computing'.lower())` `computing` `print('Com pu ting' .lower())` `com pu ting`

String Method	Argument(s)	Description	Example
`split()`	None	Returns a `list` of `str`s that are separated by one or more spaces in the original `str`.	`print('Hi, World!' .split())` → `['Hi,', 'World!']` `print('C S'.split())` → `['C', 'S']`
`split()`	A `str`	Returns a `list` of `str`s that are separated by the argument in the original `str`.	`print('Hi, World' .split(','))` → `['Hi', ' World!']` `print('C  S'.split(' '))` → `['C', '', 'S']`
`startswith()`	A `str`	Returns a `bool` that indicates whether the `str` starts with the given argument.	`print('CS2024' .startswith('CS'))` → `True` `print('CS2024' .startswith('24'))` → `False`
`upper()`	None	Returns a `str` containing every character of the original `str` converted to upper case.	`print('Computing'.upper())` → `COMPUTING` `print('Com pu ting' .upper())` → `COM PU TING`

Table 4.38 Built-in string methods

4.11.5 Formatting Strings

We have already seen how to use f-strings as well as type casting and the concatenation operator to produce `str`s that include other values and follow a desired format. For instance, suppose we wish to report the values assigned to variables named `student` and `year` with some short descriptive text. We can use f-strings like this:

```
student = "Bala"
year = 2024
print(f"student is {student}, year is {year}.")
```

```
student is Bala, year is 2024.
```

Figure 4.105 Using f-strings to format `str`s

We can also use type casting and the concatenation operator like this:

```
student = "Bala"
year = 2024
print("student is " + student + ", year is " + str(year) + ".")
```

```
student is Bala, year is 2024.
```

Figure 4.106 Using type casting and the concatenation operator to format strs

In this section, we will cover how to produce the same output using the `str.format()` method like this:

```
student = "Bala"
year = 2024
print("student is {}, year is {}.".format(student, year))
```

```
student is Bala, year is 2024.
```

Figure 4.107 Using the `str.format()` method to format strs

Notice that the output of `str.format()` replaces each set of curly braces `{}` in the original `str` with one of the provided arguments. As with f-strings, each set of curly braces is called a "replacement field" or "field", but the syntax here is different and fields for `str.format()` cannot contain arbitrary Python expressions. By default, fields are replaced with arguments in order from left to right. For instance, in Figure 4.107, the first `{}` is replaced with the first argument student while the second `{}` is replaced with the second argument `year`.

To change the order of arguments used to replace each field, we specify an index for each field as follows:

```
student = "Bala"
year = 2024
print("year is {1}, student is {0}.".format(student, year))
```

```
year is 2024, student is Bala.
```

Figure 4.108 Specifying an index for each field

Just like indices for `lists` and `strs`, indices for arguments start from 0 and not 1. Hence, the field of `{0}` specifies an index of 0 and is replaced by the first argument `student`. The field of `{1}` specifies an index of 1 and is replaced by the second argument `year`.

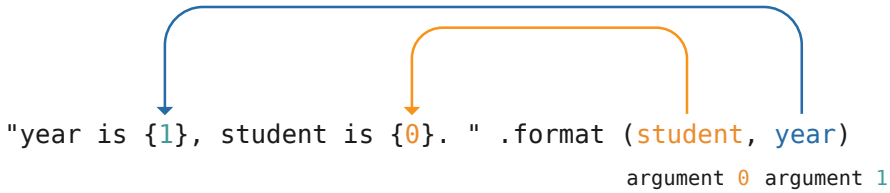


Figure 4.109 Arguments are matched to fields based on index

Note that multiple fields may specify the same index and that it is not compulsory for all arguments to be included in the output.

```
student = "Bala"
year = 2024
print("{0}, {0}, {0}".format(student, year))
```

```
Bala, Bala, Bala
```

```
print("{1}, {0}, {1}".format(student, year))
```

```
2024, Bala, 2024
```

Figure 4.110 Examples of using fields with specified indices

Also note that fields with specified indices and fields without specified indices cannot be used together.

```
print("{}, {1}, {0}".format(student, year))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[10], line 1
----> 1 print("{}, {1}, {0}".format(student, year))

ValueError: cannot switch from automatic field numbering to manual field specification
```

Figure 4.111 Fields with and without specified indices cannot be used together

As with f-strings, to "escape" the curly braces so that they are not treated as a field, use double braces so that it looks like "{{" or "}}":

```
student = "Bala"
year = 2024
print("student is {}, ignore {{}}, year is {}".format(student, year))
```

```
student is Bala, ignore {}, year is 2024
```

Figure 4.112 Using double curly braces to prevent them from being treated as a field

In summary, the syntax for a `str.format()` replacement field is as follows:

Syntax 4.9 Replacement field (str.format method)

```
{}
{index}
```

Figure 4.113 Syntax for replacement fields (`str.format` method)

QUICK CHECK 4.11

1. State whether each of the expressions below evaluate to `True` or `False`.

a) `"b" not in "abc"`
b) `"c" in "abc"`
c) `"d" in "abc"`

2. Predict the output of the following code:

```
example = "x"
example += 'y'
example *= 2
print(example)
```

3. Predict the output of the following code:

```
word1 = 'more'
word2 = 'or'
word3 = 'less'
result = (2 * word1) + (word2 + word3) * 2
print(result)
```

Hint: The standard order of operations and meaning of brackets in mathematics still apply.

4. Examine the following program:

```
s = "x\nx"
s = 2 * s
s = s + "\n"
s = s * 2
print(s)
```

How many non-empty lines of output will there be?

5. A string satisfies the format of a postal code if has exactly 6 characters, all of which are digits. Write a program to solve the following postal code format checking problem:

Input	Output
• *postal_code*: a string	• Whether *postal_code* satisfies the format of a postal code (i.e., True or False)

Table 4.39 Input and output requirements for postal code format checking problem

For instance, if the input is “012345”, the output should be “True”. However, if the input is “12345”, the output should be “False”.

6. A string satisfies the format of a postal code if has exactly 6 characters, all of which are digits.

Write a program to solve the following boxed message problem:

Input	Output
• *message*: a string	• Text-based box made out of “*” characters surrounding the contents of *message*

Table 4.40 Input and output requirements for boxed message problem

For instance, if the input is “Hello, World!”. the output should be:

```
***************
*Hello, World!*
***************
```

4.12 Lists

LEARNING OUTCOMES

2.3.9 *Use list values with appropriate operators, built-in functions and methods (limited to those mentioned in the Quick Reference Guide) to perform:*
- *Concatenation and repetition*
- *Extraction of single items and subset of items (i.e., indexing and slicing)*
- *Testing of whether an item is in the list*
- *Calculation of length*
- *Calculation of sum, minimum value and maximum value (provided the list items are all integer or floating-point values)*

A `list` is an ordered sequence of changeable values.

`lists` are useful to keep and manipulate multiple values without prior knowledge of the total number of values. This is an important advantage over using multiple variables, where the number of variables must be determined ahead of time.

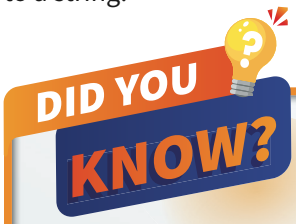
4.12.1 List Literals

`list` literals are written as a comma-separated list of values enclosed by square brackets.

```
print([1, 2, 3])
[1, 2, 3]
print(["Mixing types in a list like this is bad.", 2024, False])
["Mixing types in a list like this is bad.", 2024, False]
print([])
[]
```

Figure 4.114 Examples of valid `list` literals

Note how a `list` is printed in a format like its literal form. The same format is used when a `list` is converted to a string.



A simple `list` is sometimes called a one-dimensional "array". Technically, an array requires all its items to have the same size and be arranged consecutively in memory. Although a `list` may contain values of different types and sizes, internally it tracks those values using an array of memory addresses that all have the same size, so this description is acceptable.

Besides built-in `lists`, Python offers more traditional arrays that contain values of a fixed type via the `array` module. Use of the `array` module is not covered in this textbook.

In particular, the list of values in the brackets can be empty and the `[]` literal is often used to create a new empty `list`.

Syntax 4.10 list Literals

```
[]
[value_1]
[value_1, value_2]
```

Figure 4.115 Syntax for `list` literals

4.12.2 List Operators

4.12.2.1 Comparison

Many of the same operators that work on strings also work on `lists`, such as the comparison operators in Table 4.41. When comparing `lists`, the results are determined by comparing the first pair of differing values. If no difference is found, the comparison is based on the lengths of the `lists`.

Operator	Name	Operand(s)	Description	Example
<	Less than	Two `lists`	Returns `True` if the `list` on the left is less than the `list` on the right and `False` otherwise.	`x = [2, 0, 2, 4]` `y = [2, 0, 3, 0]` `z = [2, 0]` `print(x < x)` `False` `print(x < y)` `True` `print(x < z)` `False` *Note: < also works with numbers and `strs`. See Table 4.18 and Table 4.33.*

Operator	Name	Operand(s)	Description	Example
<=	Less than or equal to	Two `lists`	Returns `True` if the `list` on the left is less than or equal to the `list` on the right and `False` otherwise.	`x = [2, 0, 2, 4]` `y = [2, 0, 3, 0]` `z = [2, 0]` `print(x <= x)` `True` `print(x <= y)` `True` `print(x <= z)` `False` *Note: <= also works with numbers and `strs`. See Table 4.18 and Table 4.33.*
>	Greater than	Two `lists`	Returns `True` if the `list` on the left is greater than the `list` on the right and `False` otherwise.	`x = [2, 0, 2, 4]` `y = [2, 0, 3, 0]` `z = [2, 0]` `print(x > x)` `False` `print(x > y)` `False` `print(x > z)` `True` *Note: > also works with numbers and `strs`. See Table 4.18 and Table 4.33.*
>=	Greater than or equal to	Two `lists`	Returns `True` if the `list` on the left is greater than or equal to the `list` on the right and `False` otherwise.	`x = [2, 0, 2, 4]` `y = [2, 0, 3, 0]` `z = [2, 0]` `print(x >= x)` `True` `print(x >= y)` `False` `print(x >= z)` `True` *Note: >= also works with numbers and `strs`. See Table 4.18 and Table 4.33.*

Table 4.41 Comparison operators for `lists`

Note that since the results are obtained by comparing corresponding items in the two `lists`, in general the data types of items in the two `lists` must be compatible for performing comparisons or an error will occur:

```
x = [2, '0', 20]
y = [2, '0', 30]
z = [2, '0', '30']
print(x <= y)      # OK since types match
```

```
True
```

```
print(x <= z)      # Error since types at index 2 are not compatible
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[3], line 1
----> 1 print(x <= z)      # Error since types at index 2 are not compatible

TypeError: '<=' not supported between instances of 'int' and 'str'
```

Figure 4.116 Types of `list` values must be compatible to perform comparison

4.12.2.2 Concatenation and Repetition

As with strings, the + and * operators can be used to join and repeat `lists`:

Operator	Name	Operand(s)	Description	Example
+	Concatenation	Two `lists`	Returns a new `list` where the two `lists` are joined together	`print([1965] + [2021])` `True` *Note: + is also used for addition. See Table 4.19.* *Note: + also works with `strs`. See Table 4.34*
*	Repetition	A `list` and an `int`, in any order	Returns a new `list` where the given `list` is repeated a given number of times	`print([0, 1] * 3)` `[0, 1, 0, 1, 0, 1]` `print(3 * [1, 2])` `[1, 2, 1, 2, 1, 2]` *Note: * is also used for multiplication. See Table 4.19.* *Note: * also works with `strs`. See Table 4.34.*

Table 4.42 Concatenation and repetition operators for `lists`

4.12.2.3 Indexing and Slicing

The indexing and slicing operators also work with `lists`:

Operator	Name	Operand(s)	Description	Example
`[i]`	Indexing	A `list` and an `int`	Returns the value in the `i`-th position of the `list`	`l = [1, 9, 6, 5, 0]` `print(l[0])` `1` `print(l[3])` `5` `print(l[-1])` `0` *Note: The indexing operator also works with `strs` and `dicts`. See Table 4.35 and Table 4.46.*
`[a:b]`	Slicing	A `list` and two optional `ints`	Returns a `list` starting from the value at index a up to but not including the value at index b; a is treated as 0 if omitted and b is treated as the length of the sequence if omitted	`l = [1, 9, 6, 5, 0]` `print(l[1:4])` `[9, 6, 5]` `print(l[:2])` `[1, 9]` `print(l[2:])` `[6, 5, 0]` *Note: The slicing operator also works with `strs`. See Table 4.35.*
`[a:b:c]`	Slicing with step	A `list` and three optional `ints`	Returns a `list` starting from the value at index a up to but not including the value at index b, in increments of c; a is treated as 0 if omitted, b is treated as the length of the sequence if omitted and c is treated as 1 if omitted.	`l = [1, 9, 6, 5, 0]` `print(l[1:4:2])` `[9, 5]` `print(l[:4:2])` `[1, 6]` `print(l[1::2])` `[9, 5]` `print(l[1:4:])` `[9, 6, 5]` *Note: The slicing with step operator also works with `strs`. See Table 4.35.*

Table 4.43 Indexing and slicing operators for `lists`

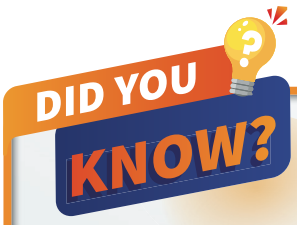
Unlike `strs` which are immutable, with `lists` we can use the indexing operator on the left-hand side of an assignment statement to change the value stored in a `list`:

Notice how the third value (with index 2) of the `list` changed from 72.9 to 50.0.

```
scores = [85.0, 88.1, 72.9, 63.4]
scores[2] = 50.0
print(scores)
```

```
[85.0, 88.1, 50.0, 63.4]
```

Figure 4.117 Changing a value in a `list` using the indexing operator



Be aware that changing the contents of a `list` may have unintended consequences if multiple variables refer to the same `list`. Recall from section 4.5.2 that variables in Python behave like sticky notes attached to memory addresses. If two variables x and y refer to the same `list`, then changing the `list` using x will also affect y:

```
x = [1965]
y = x          # x and y refer to the same mutable list
print(x)
print(y)
```

```
[1965]
[1965]
```

```
x[0] = 2024     # list is changed using x
print(x)
print(y)        # y reflects the change
```

```
[2024]
[2024]
```

Figure 4.118 Changes to a mutable value can affect multiple variables that refer to it

Figure 4.119 Conceptual diagram that shows x and y referring to the same modified value

This is different from having multiple variables refer to `ints`, `floats` or `strs` as these values are immutable. Suppose two variables x and y refer to the same immutable value (such as an `int`). If x gets replaced with a different value, that value will be separate and have a different memory address from the original, so y is unaffected:

```
x = 1965
y = x          # x and y refer to the same immutable int
print(x)
print(y)
```

```
1965
1965
```

```
x = 2024        # x is replaced with a new, separate int
print(x)
print(y)        # y is unaffected
```

```
2024
1965
```

Figure 4.120 Immutable values can only be replaced and not changed

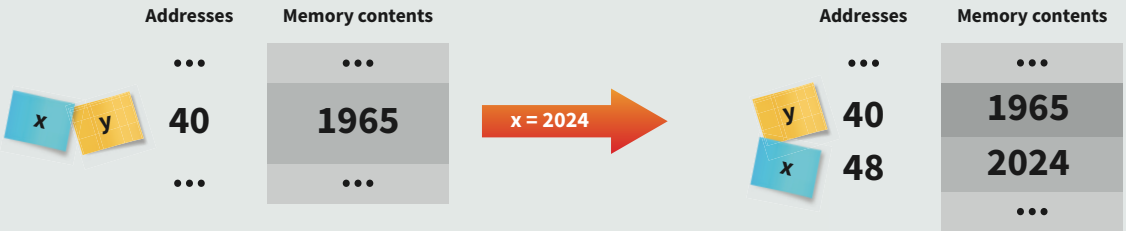


Figure 4.121 Conceptual diagram that shows x referring to a new value

4.12.2.4 Membership

We can also use the membership operator "in" to test if an item appears within a `list`:

Operator	Name	Operand(s)	Description	Example
`in`	Membership	Any value and a `list`	Returns `True` if the value on the left can be found in the `list` on the right and `False` otherwise.	`print(0 in [2, 0, 1])` `True` `print(3 in [2, 0, 1])` `False` *Note: `in` also works with `strs` and `dicts`. See Table 4.36 and Table 4.47.*
`not in`	Non-membership	Any value and a `list`	Returns `True` if the value on the left cannot be found in the `list` on the right and `False` otherwise.	`print(0 not in [2, 0, 1])` `False` `print(3 not in [2, 0, 1])` `True` *Note: `not in` also works with `strs` and `dicts`. See Table 4.36 and Table 4.47.*

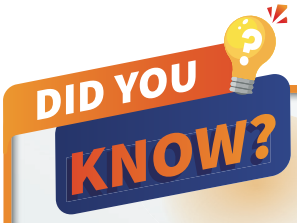
Table 4.44 Membership operators for `lists`

4.12.3 List Functions

Table 4.45 summarises some common `list` functions in Python:

Function	Argument(s)	Description	Example
`len()`	A `list`	Returns an `int` equal to the number of values in the given `list`.	`print(len([2, -2, 4]))` `3` `print(len([]))` `0` *Note: `len()` also works with `strs` and `dicts`. See Table 4.37 and Table 4.48.*
`list()`	Any value that can be converted to a `list`	Returns a `list` representation of the given argument (if possible). For `strs`, returns a `list` of individual characters. For ranges, returns a `list` of `ints`. For `dicts`, returns a `list` of keys.	`print(list('App'))` `['A', 'p', 'p']` `r = range(5)` `print(list(r))` `[0, 1, 2, 3, 4]` `d = {'x': 2, 'y': 0}` `print(list(d))` `['x', 'y']`
`max()`	A `list` of numbers (`int` or `float`)	Returns the maximum value out of the values in the given `list`.	`print(max([2, -2, 4]))` `4` *Note: `max()` also works with multiple arguments. See Table 4.21.*
`min()`	A `list` of numbers (`int` or `float`)	Returns the minimum value out of the values in the given `list`.	`print(min([2, -2, 4]))` `-2` *Note: `min()` also works with multiple arguments. See Table 4.21.*
`sum()`	A `list` of numbers (`int` or `float`)	Returns the sum of values in the given `list`.	`print(sum([2, -2, 4]))` `4`

Table 4.45 Built-in `list` functions



`lists` are also objects and one of the methods they have is `append()`, which accepts a value and adds it to the end of the `list`. It is often used to build up a `list` item by item:

```
years = []
years.append(1965)
years.append(2024)
print(years)
```

```
[1965, 2024]
```

Figure 4.122 Using the `append()` method to build up a `list`

Note that `append()` modifies the `list` directly and does not have a return value. Since it modifies the `list`, other variables that refer to the same `list` will also be affected, as mentioned in section 4.12.2:

```
years = [1965, 2024]
years_2 = years
years.append(2030)    # Original list is changed
print(years)
print(years_2)        # years_2 reflects the change
```

```
[1965, 2024, 2030]
[1965, 2024, 2030]
```

Figure 4.123 The `append()` method modifies the `list`

An alternative to using `append()` is using the concatenation operator to join the original `list` to a `list` containing just the new item:

```
years = []
years = years + [1965]
years = years + [2024]
print(years)
```

```
[1965, 2024]
```

Figure 4.124 Alternative method of building up a `list` by concatenation

Note that each use of the concatenation operator creates a new `list` and this process is slower than using `append()`. As the new `list` is separate from the original `list`, other variables that refer to the original `list` are unaffected:

```
years = [1965, 2024]
years_2 = years
years = years + [2030]       # years is replaced with a new, separate list
print(years)
print(years_2)               # years_2 is unaffected
```

```
[1965, 2024, 2030]
[1965, 2024]
```

Figure 4.125 The concatenation operator creates a new `list` each time it is used

QUICK CHECK 4.12

1. State whether each statement is true or false.

a) In general, the + operator requires the values on its left and right sides to have the same type (except mixing `int`s and `float`s are allowed because they both represent numbers).

b) The code `[] + []` is valid and the result is `[]`.

c) The code `1 + []` is valid and the result is `[1]`.

d) The code `[1] + 1` is valid and the result is `[1]`.

e) The code `"A" + "B"` is valid and the result is `"AB"`.

f) The code `["A"] + ["B"]` is valid and the result is `["A", "B"]`.

2. It can be difficult to distinguish between `list` literals, indexing operators and the slicing operators as they all use square brackets in their syntax.

For each of the following programs, identify all the line numbers where there are:

i. List literals

ii. Uses of an indexing operator

iii. Uses of a slicing operator

a)

```
1 choices = [1965, 2017, 2024, 2030]
2 year = choices[2]
3 print(year)
```

b)

```
1 year = [1965, 2017, 2024, 2030][2]
2 print(year)
```

c)

```
1 numbers = [1, 9, 6, 5, 0]
2 numbers = numbers + [2]
3 midpoint = len(numbers) // 2
4 print(numbers[midpoint])
5 half1 = numbers[:midpoint]
6 half2 = numbers[midpoint:]
7 print(half2 + half1)
```

3. What is the output of the following code?

```
numbers = [1]
numbers += [2]
print(numbers)
```

A) `[1, [2]]`
B) `[1, 2]`
C) `[3]`
D) `3`

4. Predict the output of the following code:

```
sales = []
sales += [44]
sales += [55]
print(sales)
```

5. Predict the output of the following code:

```
sequence = [1, 4]
sequence *= 3
print(sequence)
```

QUICK CHECK 4.12

6. Predict the output of the following code:

```
scores = [92, 97, -1, 93, 96, -1]
scores[2] = 99
scores[5] = 50
print(scores)
```

7. Predict the output of the following code:

```
name1 = 'Alex'
name2 = 'Bala'
name3 = 'Siti'
names = [name1, name2, name3]
print('name2' in names)
```

4.13 Dictionaries

LEARNING OUTCOMES

2.3.10 *Use dictionary values with appropriate operators to perform dictionary insertion, query, lookup and deletion.*

A **dictionary** (or `dict`) is an unordered collection of key-value pairs.

`dicts` are useful for problems that involve pairs of values where it is useful to quickly look up the second value of a pair given only the first value of pair, which is called the **key**.

The name "dictionary" is based on the metaphor of physical dictionaries, which are used to look up definitions (i.e., values) of words (i.e., keys).

KEY TERMS

Dictionary (dict)
A data type to represent an unordered collection of key-value pairs

Key
A unique value that is used to look up the corresponding value in a dictionary

However, `dicts` are also useful for other problems that have nothing to do with words or definitions. For instance, they can be used to look up the name of a user given their email address or to look up the price of a item given its product code.

The keys and values of a dictionary satisfy certain requirements and properties:

1. Keys in a dictionary must be unique
2. Conversely, the values in a dictionary do not need to be unique
3. Keys must be immutable (i.e., `lists` cannot be used as keys)
4. Each key in a dictionary is paired with exactly one value

4.13.1 Dictionary Literals

`dict` literals can be written as a comma-separated list of key-value pairs in the form "key: value" enclosed by curly brackets:

```
empty_dict = {}
english_names = {1: 'one', 2: 'two', 3: 'three'}
print(english_names)
```

```
{1: 'one', 2: 'two', 3: 'three'}
```

Figure 4.126 Examples of valid dictionary literals

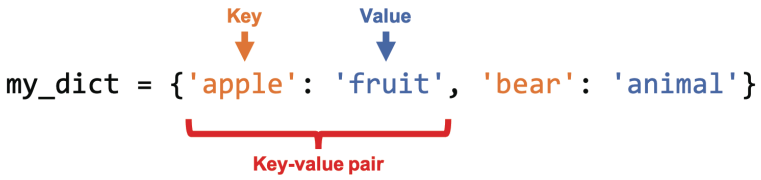


Figure 4.127 Labelled example of a dictionary literal

Note that like a `list`, a `dict` is printed in a format like its literal form.

The `{}` literal is often used to create a new empty `dict`.

Syntax 4.11 dict Literals

```
{}
{key_1: value_1}
{key_1: value_1}, {key_2: value_2}
```

Figure 4.128 Syntax for `dict` literals

4.13.2 Dictionary Operators

4.13.2.1 Indexing

Looking up the value associated with a key uses the indexing operator:

Operator	Name	Operand(s)	Description	Example
`[key]`	Indexing	A `dict` and an immutable value	Returns value associated with given key in the `dict`.	`d = {1: 9, 6: 5}` `print(d[1])` 9 `print(d[6])` 5 *Note: The indexing operator also works with `str`s and `list`s. See Table 4.35 and Table 4.43.*

Table 4.46 Indexing operator for `dict`s

To set or change the value associated with a key, use the indexing operator on the left-hand side of an assignment statement:

```
scores = {'Alex': 176, 'Bala': 180, 'Siti': 177}

# Insert score for new player John
scores['John'] = 174
print(scores)
```

```
{'Alex': 176, 'Bala': 180, 'Siti': 177, 'John': 174}
```

```
# Update Alex's score
scores['Alex'] = 178
print(scores)
```

```
{'Alex': 178, 'Bala': 180, 'Siti': 177, 'John': 174}
```

Figure 4.129 Setting and changing dictionary values

To remove a key-value pair, use a `del` statement with the indexing operator:

```
scores = {'Alex': 176, 'Bala': 180, 'Siti': 177}

# Remove Bala's score
del scores['Bala']
print(scores)
```

```
{'Alex': 176, 'Siti': 177}
```

Figure 4.130 Deleting a key-value pair from a dictionary

4.13.2.2 Membership

To test whether a key exists in a dictionary, use the "in" membership operator.

Operator	Name	Operand(s)	Description	Example
`in`	Membership	Any value and a `dict`	Returns `True` if the value on the left is a key in the `dict` on the right and `False` otherwise.	`d = {'x': 2, 'y': 0}` `print('y' in d)` `True` `print(0 in d)` `False` *Note: `in` also works with `str`s and `list`s. See Table 4.36 and Table 4.44.*

Operator	Name	Operand(s)	Description	Example
`not in`	Non-Membership	Any value and a `dict`	Returns `True` if the value on the left is not a key in the `dict` on the right and `False` otherwise.	`d = {'x': 2, 'y': 0}` `print('y' not in d)` `True` `print(0 not in d)` `True` *Note: `not in` also works with `strs` and `lists`. See Table 4.36 and Table 4.44.*

Table 4.47 Membership operators for `dicts`

4.13.3 Dictionary Functions

Table 4.48 summarises how the `len()` function can be used to determine the number of key-value pairs in a dictionary:

Funtion	Argument	Returns	Examples
`len()`	`dict`	Returns an `int` equal to the number of keys in the given `dict`.	`d = {'x': 2, 'y': 0}` `print(len(d))` `2` `print(len({}))` `0` *Note: `len()` also works with `strs` and `lists`. See Table 4.37 and Table 4.45.*

Table 4.48 Built-in dictionary functions

sstctf{1_h0p3_y0u_summ4ris3d_th1s_f0r_ur_CP+_E0Y}

QUICK CHECK 4.13

1. If d is a dictionary, what does "`d[2]`" evaluate to?

A) The second item stored in d.
B) The third item stored in d.
C) The key associated with the value 2 in d.
D) The value associated with the key 2 in d.

2. What is the output of the following code?

```
number_pairs = {10: 22, 15: 20, 22: 15, 16: 10, 20: 16}
print(number_pairs[20])
```

A) 15
B) 16
C) 20
D) 22

3. Predict the output of the following code:

```
data = {}
data['ID1'] = ['Alice', 1.55]
data['ID2'] = ['Bob', 1.67]
print(data['ID1'])
```

4. Predict the output of the following code:

```
status = {1: True, 2: False}
print(True in status)
```

5. Predict the output of the following code:

```
codebook = {13: 'A', 29: 'C', 55: 'G', 57: 'I', 71: 'M'}
codes = [71, 13, 55, 57, 29]
message = codebook[codes[0]]
message += codebook[codes[1]]
message += codebook[codes[2]]
message += codebook[codes[3]]
message += codebook[codes[4]]
print(message)
```

4.14 Control Flow

LEARNING OUTCOMES

2.2.1 *Interpret flowcharts to understand the sequence, selection and iteration constructs.*
2.3.11 *Use the if, elif and else keywords to implement selection constructs.*
2.3.12 *Use the for and while keywords to implement iteration constructs.*

Control flow refers to the order in which the instructions of a program are run. In this section, we will explore flowcharts and use them to understand the control flow commands in Python.

4.14.1 Flowcharts

The algorithm below shows how to calculate the remainder when a positive integer *dividend* is divided by another positive integer *divisor*:

1. Get user to input positive integers *dividend* and *divisor*.
2. If *dividend* is less than *divisor*, go to Step 4. Otherwise, continue to Step 3.
3. Subtract *divisor* from *dividend* and store the difference in *dividend*, then go back to Step 2.
4. Output *dividend* as the final answer.

This same algorithm can also be represented using the **flowchart** as shown in Figure 4.131.

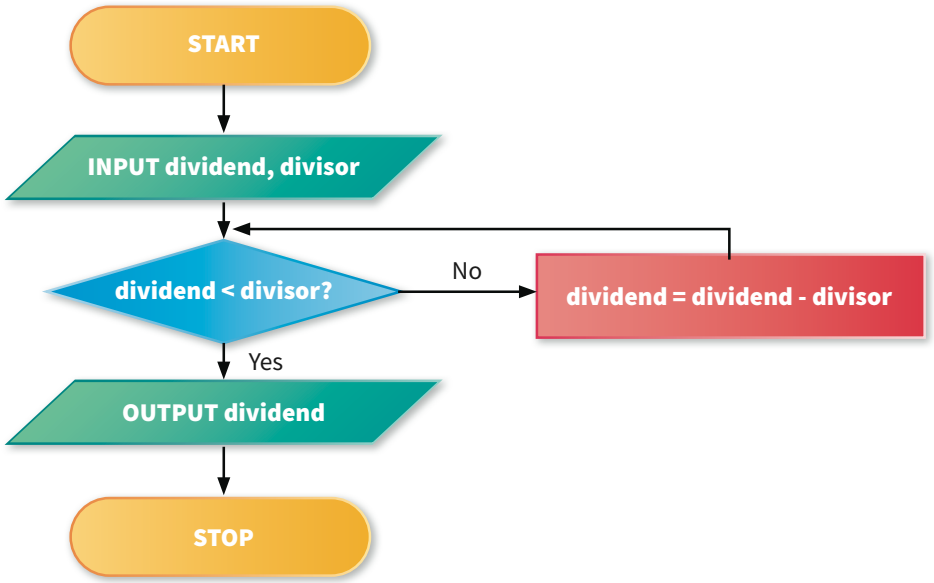
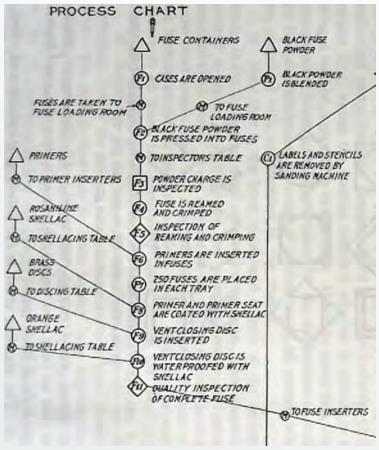


Figure 4.131 Remainder algorithm represented in a flowchart

As can be seen, a flowchart is a more visual way of presenting an algorithm and may be easier to understand than a long series of steps. Each step is represented by a **symbol** and the order in which steps are followed is shown using **flow lines** or arrows.

DID YOU KNOW?

The earliest forms of flowcharts were introduced by engineers Frank and Lillian Gilbreth in 1921 to visualise processes used in industrial production, sales and finance.



There are four standard symbols used in flowcharts:

1. Terminator

2. Data

3. Decision

4. Process

KEY TERMS

Control flow
The order in which instructions of a program are run

Flow line
An arrow that indicates the order in which steps should be followed

Flowchart
A visual presentation of an algorithm using symbols to show the flow of a sequence of steps

Symbol (flowchart)
A special shape used to represent an action or step in a flowchart

4.14.1.1 Terminator Symbol

The terminator symbol is a rectangle with rounded corners, or a "pill" shape.

Figure 4.132 The terminator symbol

It represents the beginning or end of a set of steps and is usually labelled with either START or END/STOP as shown in Figure 4.133.

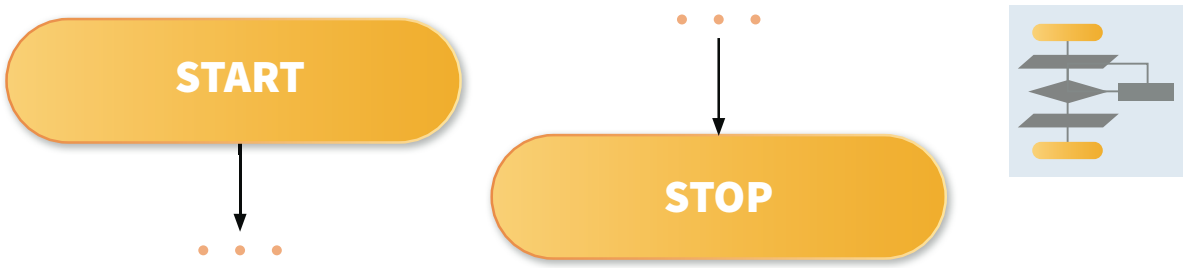


Figure 4.133 Examples of how the terminator symbol is used

4.14.1.2 Data Symbol

The data symbol is a parallelogram.

Figure 4.134 The data symbol

It represents either receiving input data from outside the algorithm (usually labelled with INPUT) or producing output from within the algorithm (usually labelled with OUTPUT).

An example of each use is shown in Figure 4.135.

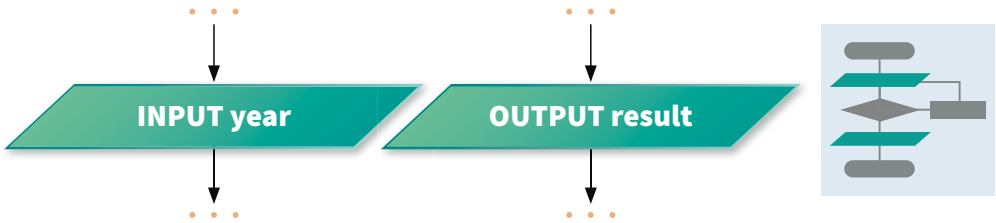


Figure 4.135 Examples of how the data symbol is used

4.14.1.3 Decision Symbol

The decision symbol is a diamond.

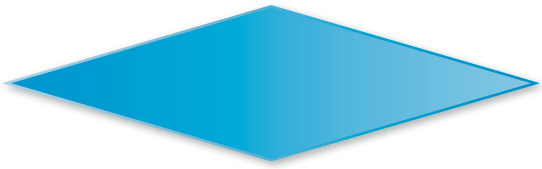
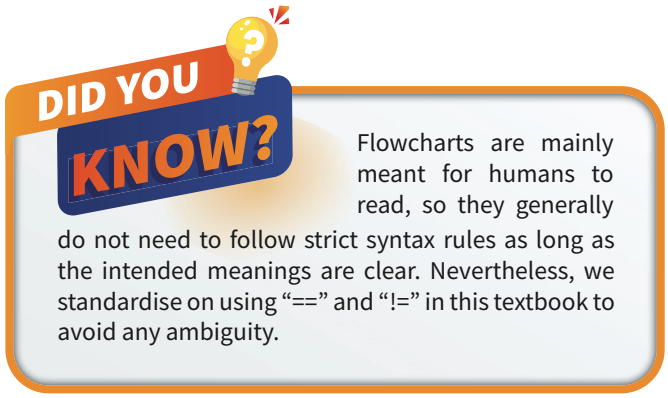


Figure 4.136 The decision symbol

It represents a step involving a question. The outgoing arrows represent the possible outcomes to the question and are usually labelled "Yes" and "No". There may be two or three outgoing arrows depending on the number of possible outcomes. Only one of these outgoing arrows should be followed when performing the algorithm.

Note that the question used in a decision symbol may involve checking if two values are equal or not equal. We typically use the Python operators "==" and "!=" to represent such checks. For example, to check if the variables x and y are equal, we would ask "Is x == y?" Conversely, to check if the variables x and y are not equal, we would ask "Is x != y?"

Some examples of the decision symbol are shown in Figure 4.137.



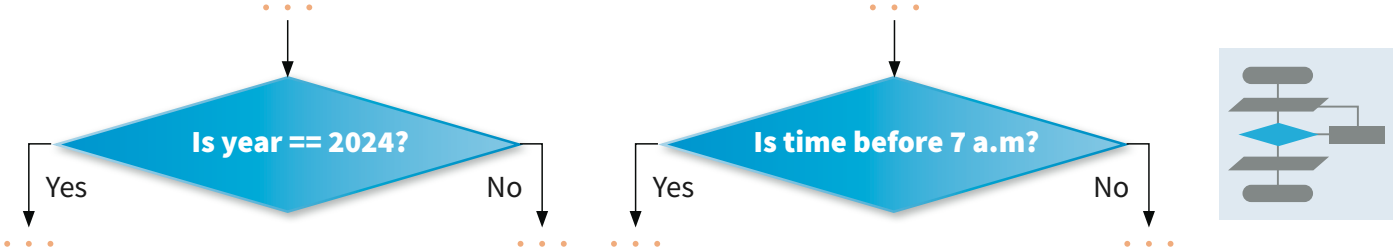


Figure 4.137 Examples of how the decision symbol is used

4.14.1.4 Process Symbol

The process symbol is a rectangle.

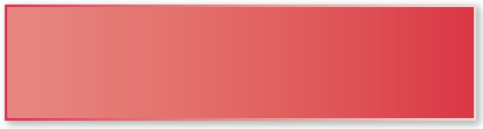


Figure 4.138 The process symbol

It represents a step involving an action or operation. This usually involves changing the value of a variable or performing more complex actions, as shown in Figure 4.139.

Figure 4.139 Examples of how the process symbol is used

4.14.2 Constructs

Flowchart symbols must be connected using flow lines or arrows to indicate the order in which they should be followed. The flow lines should also follow the sequence, selection and iteration **constructs** (which will be discussed in the following sections) to avoid ending up with overly complicated flowcharts that are difficult to read or understand. Each of these constructs are formulated to have exactly one entry point and one exit point.

Using these constructs will also ensure that the flowcharts can be translated into Python, which is a **structured programming language** that does not allow jumping to arbitrary lines of code when the program is running.

4.14.2.1 Sequence

The most straightforward way to connect multiple symbols or constructs is to link them into a chain with flow lines pointed in the same direction so that there is only one entry point and one exit point. This is called a **sequence construct** and it is used to perform multiple instructions in a fixed order.

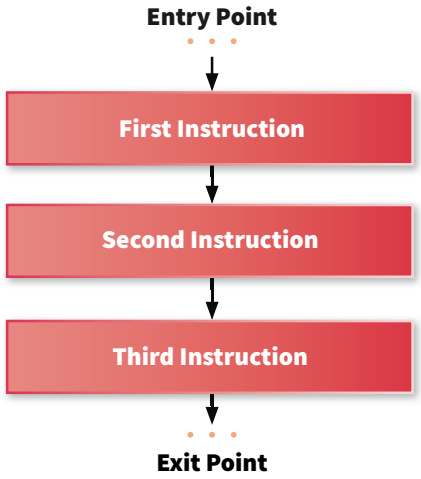


Figure 4.140 Example of a sequence construct

KEY TERMS

Construct
An algorithm structure with exactly one entry point and one exit point

Sequence construct
A construct for performing multiple instructions in a fixed order

Structured programming language
A programming language that encourages or is limited to the use of sequence, selection and iteration constructs that have exactly one entry point and one exit point

4.14.2.2 Selection

Whenever a decision symbol is used, it always has exactly one incoming flow line that serves as an entry point but may have two or more outgoing flow lines exiting from it. These outgoing flow lines lead to different **branches** or sequences of symbols and constructs. If these separate sequences merge back into a single flow line, this forms a **selection construct** with one entry point at the decision symbol and one exit point where the multiple branches merge.

We call this a selection construct as the person or computer running the flowchart has to select only one of the different branches from the decision symbol to follow. For instance, Figure 4.141 shows a selection construct with two branches (i.e., a "Yes" branch and a "No" branch).

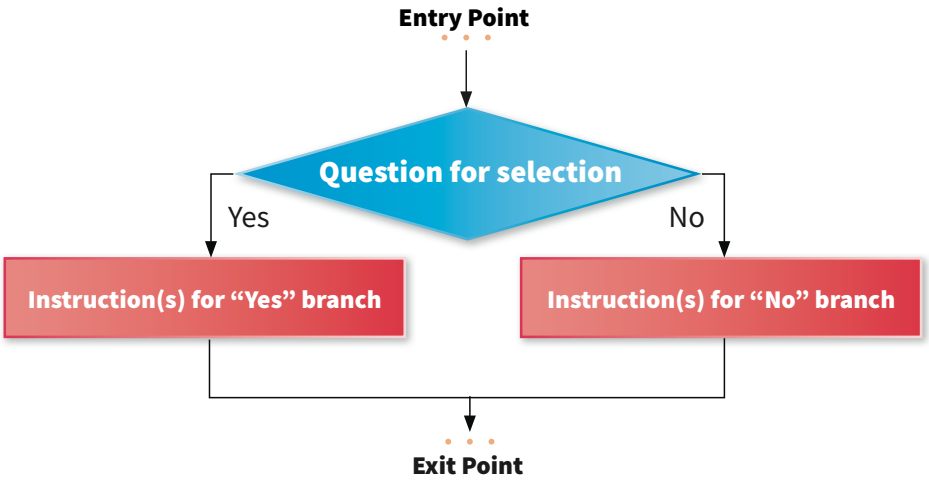


Figure 4.141 Example of a selection construct

4.14.2.3 Iteration

Sometimes, instead of having all the branches merge together, one or more of the branches may lead back to the original decision symbol, forming a closed loop. In Figure 4.142, there is one entry point at the decision symbol and one exit point at the branch that does not form a loop. This is called an **iteration construct** and it is used to repeat instructions while a particular condition is true. This construct is often used to **iterate** or repeat the instructions.

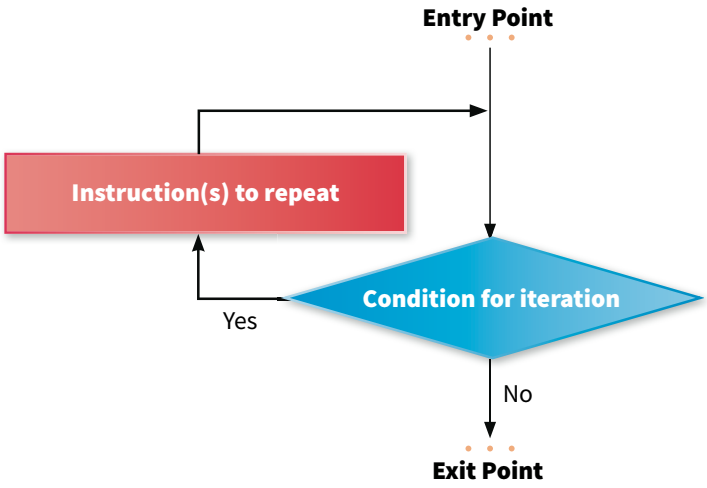


Figure 4.142 Example of iteration contstruct

KEY TERMS

Branch
A sequence of instructions that serves as one option out of two or more choices

Iterate
To repeat

Iteration construct
A construct for repeating instructions while a particular condition is true

Selection construct
A construct for choosing between two or more branches based on a particular condition

4.14.3 if-elif-else Statements

By using a selection construct, the flowchart in Figure 4.143 outputs "Yes" if the input text is the same as the special phrase "P@55w0rd", otherwise it outputs "No":

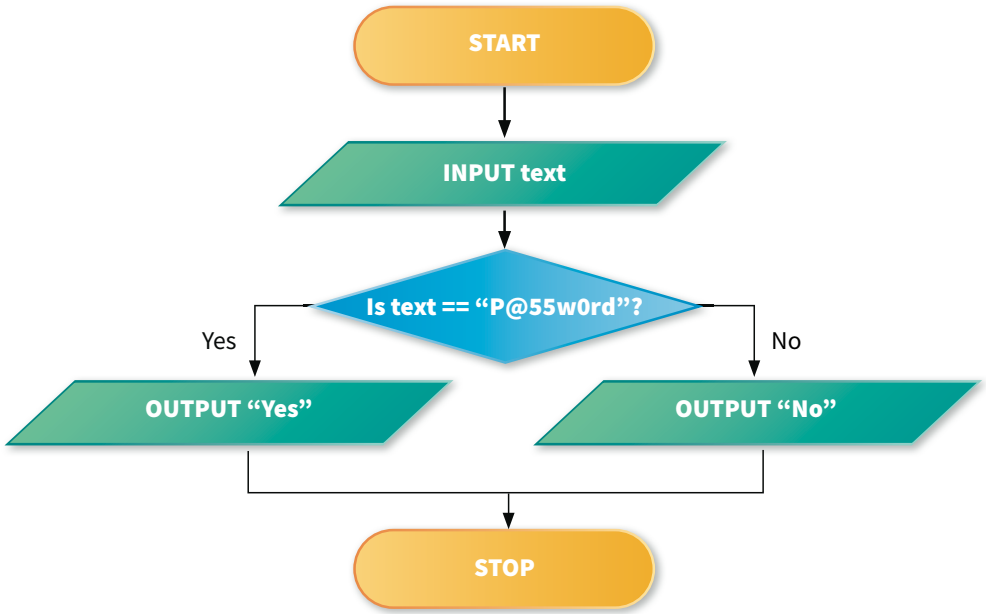


Figure 4.143 Password checker flowchart

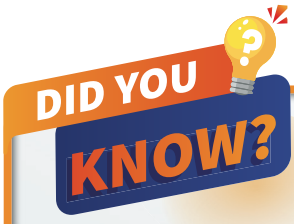
We can achieve this using the `if-else` statement in Python:

```
# Program 4.5: password.ipynb

text = input("Enter text: ")
if text == "P@55w0rd":
    print("Yes")
else:
    print("No")
```

Figure 4.144 Using the `if-else` statement

Note that some of the lines in `password.ipynb` are indented. Python uses indentation (the number of spaces at the start of each line) to determine which lines of code belong to the same branch. While this is just a good programming practice in other languages, in Python it is a requirement to indent lines of code correctly.

In this textbook, we will use four spaces to represent each level of indentation. This is the recommended and typical amount of indentation used in Python.



The programs here are just examples. Passwords are sensitive information and should never be stored on a computer in an easily readable format. You will learn more about a technique called encryption in Chapter 11 that should be used to prevent unauthorised people from reading passwords easily.

Suppose that now we want to perform more commands in each branch of `password.ipynb` and for the program to output "Goodbye" just before it ends (whether or not the input text matches the password).

The amended flowchart and Python source may look like this:

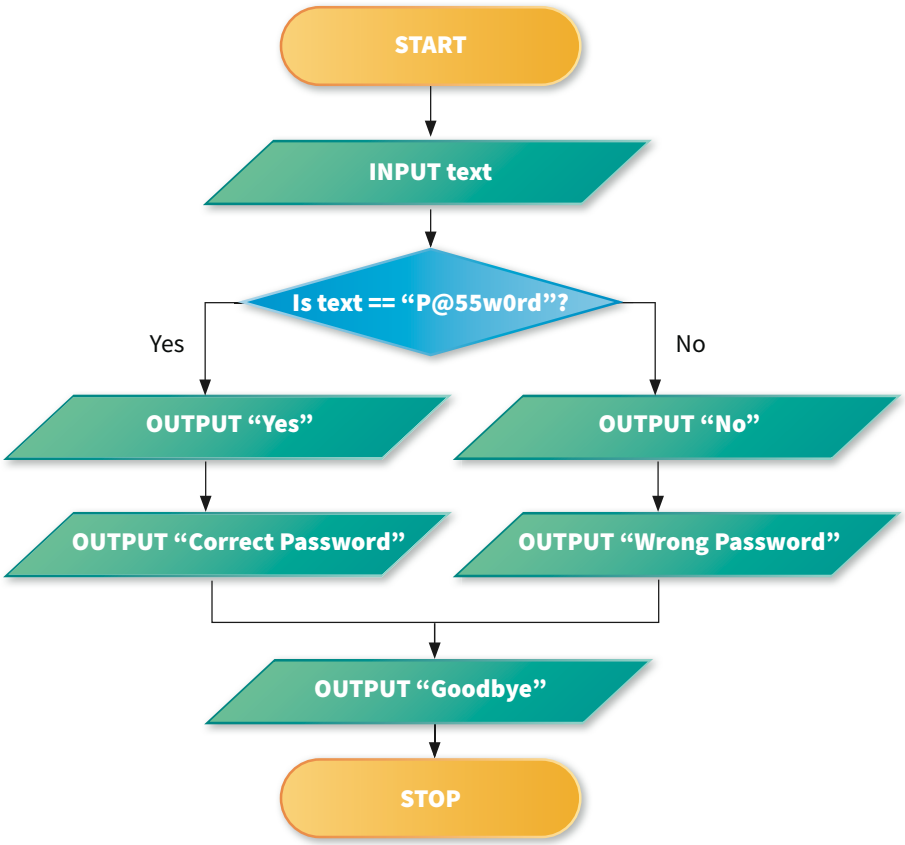


Figure 4.145 Password checker flowchart amended to include "Goodbye" output

```
   # Program 4.6: password_2.ipynb

1  text = input("Enter text: ")
2  if text == "P@55w0rd":
3      print("Yes")
4      print("Correct Password")
5  else:
6      print("No")
7      print("Wrong Password")
8  print("Goodbye")
```

Figure 4.146 Password checker program amended to include "Goodbye" output

In Figure 4.146, we see that line 8 that outputs "Goodbye" is not indented. This is how Python knows that this line is not part of the `if-else` statement and that it should be run after the `if-else` statement is completed, no matter which branch was previously followed.

It is also possible to nest an `if-else` statement inside another `if-else` statement. For instance, we might want to provide a separate error message if the text entered is blank.

One possible solution would be for the flowchart and Python source to look like this:

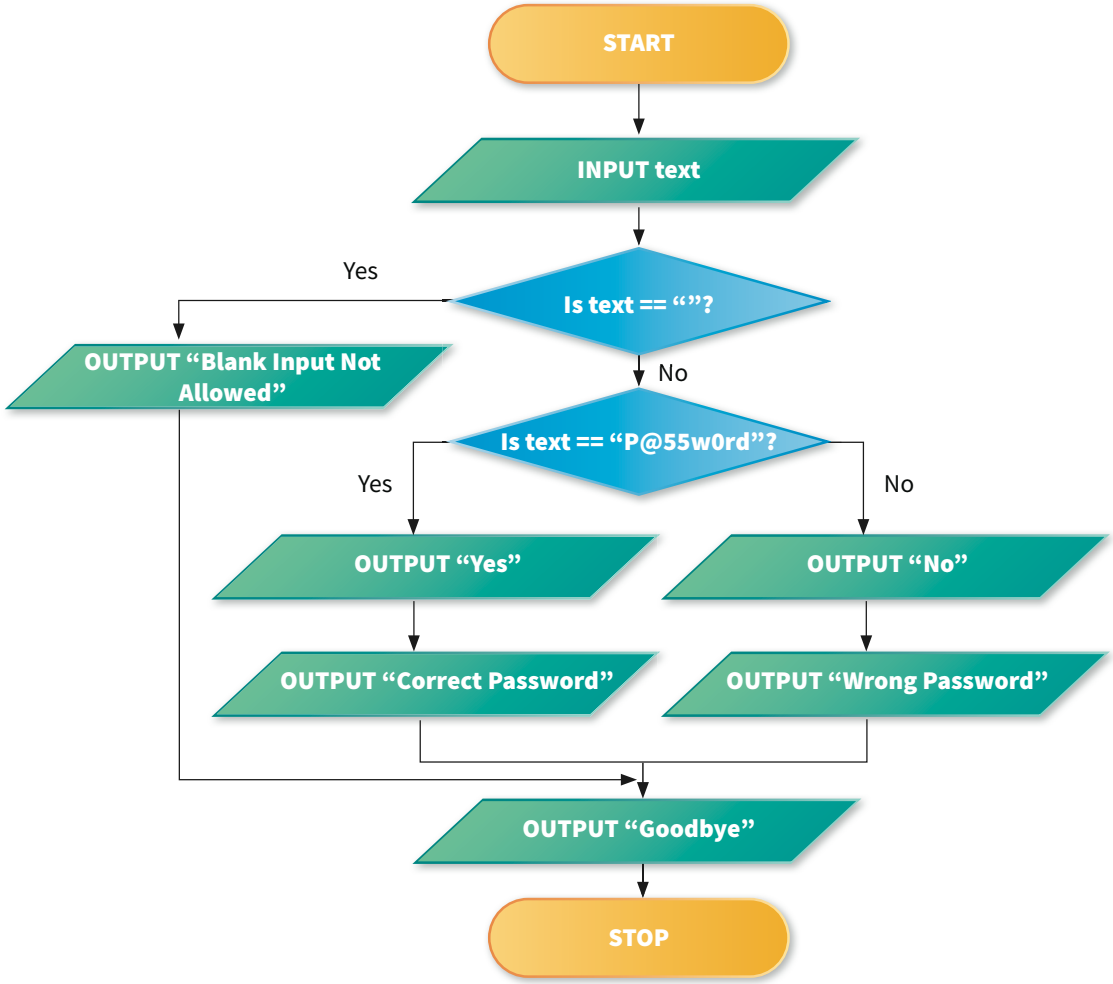


Figure 4.147 Password checker with detection of blank inp[ut added

Note that the `if-else` statements from line 5 to line 10 are indented by one additional level compared to the rest of the program. Hence, line 5 and line 8 have one level of indentation, while lines 6, 7, 9 and 10 have two levels of indentation. In Python, it is important to indicate which branch each line of code belongs to.

```
# Program 4.7: password_3.ipynb

1  text = input("Enter text: ")
2  if text == "":
3      print("Blank Input Not Allowed")
4  else:
5      if text == "P@55w0rd":
6          print("Yes")
7          print("Correct Password")
8      else:
9          print("No")
10         print("Wrong Password")
11 print("Goodbye")
```

Figure 4.148 Password checker program with detection of blank input added

Nested `if-else` statements are common. However, excessive indentation can make the code difficult to read. To avoid this, Python provides a way to combine nested `if-else` statements into a single statement so that there is no need to increase the level of indentation. For example, `password_3.ipynb` can be re-written as shown in Figure 4.149.

```
# Program 4.8: password_3_elif.ipynb

1  text = input("Enter text: ")
2  if text == "":
3      print("Blank Input Not Allowed")
4  elif text == "P@55w0rd":
5      print("Yes")
6      print("Correct Password")
7  else:
8      print("No")
9      print("Wrong Password")
10 print("Goodbye")
```

Figure 4.149 Password checker with additional indentation avoided by using elif

The `elif` keyword in line 4 of `password_3_elif.ipynb` replaces the `else` and `if` in lines 4 and 5 of the original `password_3.ipynb` and avoids additional indentation of the remaining code. When reading the source code, treat `elif` as an abbreviation for "else if". Besides these cosmetic changes, this program otherwise behaves exactly the same as the original `password_3.ipynb`.

Some algorithms do not require any instructions to be followed for the `else` portion of an `if-else` statement. The `else` keyword should then be omitted entirely. For instance, the following program tries to censor the word "Evil" if it is entered as part of a name:

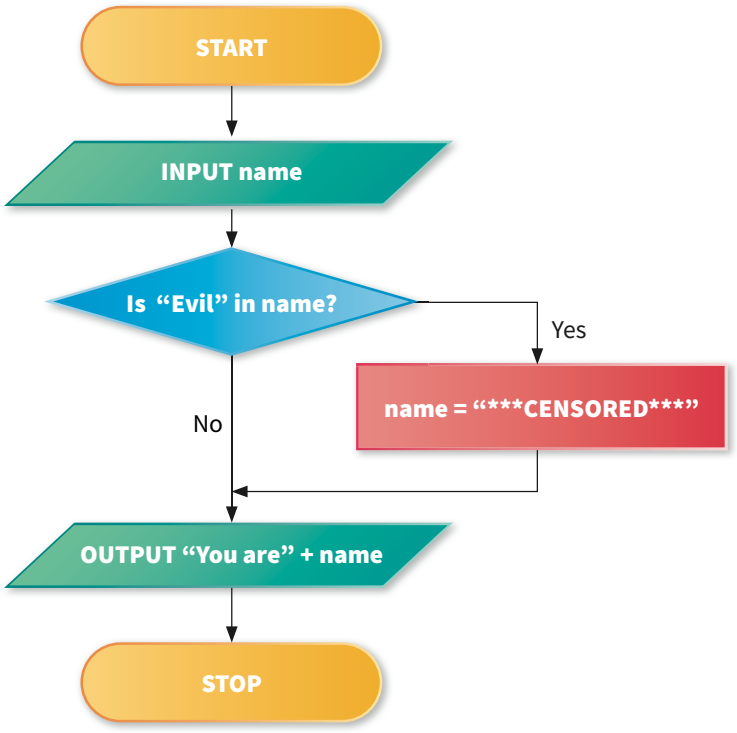


Figure 4.150 Name censor flowchart

In Figure 4.150, name has its contents replaced by a censorship notice if "Evil" is found in the name entered by the user. Otherwise, the program continues normally by outputting the name with no modification to its contents.

```
# Program 4.9: password_3_elif.ipynb

name = input("Enter name: ")
if "Evil" in name:
    name = "*** CENSORED ***"
print("You are " + name)
```

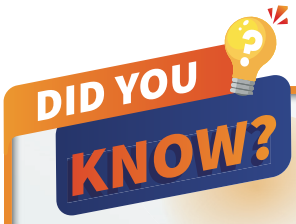
Figure 4.151 Name censor program

The syntax rules in Figure 4.152 summarise how the words and indentation for an if-elif-else statement can be arranged:

Syntax 4.12 if-elif-else Staement

```
if condition:
  commands when condition is True

if condition:
  commands when condition is True
else:
  commands when condition is False

if condition_1:
  commands when condition_1 is True
elif condition_2:
  commands when condition_1 is False and condition_2 is True
else:
  commands when condition_1 is False and condition_2 is False
```

Figure 4.152 Summary of if-elif-else statement



Run the following program that uses the `turtle` module to draw the result of rolling a 6-sided die:

```
import random
import turtle

roll = random.randint(1, 6)

turtle.penup()
turtle.goto(-50, -50)
turtle.color('black')
turtle.pendown()
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.penup()

if roll == 1:
    turtle.color('red')
    turtle.goto(0, 0)
    turtle.dot(20)
else:
    turtle.color('blue')
    turtle.goto(-25, -25)
    turtle.dot(10)
    turtle.goto(25, 25)
    turtle.dot(10)
    if roll % 2 == 1:
        turtle.goto(0, 0)
        turtle.dot(10)
    if roll > 3:
        turtle.goto(-25, 25)
        turtle.dot(10)
        turtle.goto(25, -25)
        turtle.dot(10)
        if roll == 6:
            turtle.goto(-25, 0)
            turtle.dot(10)
            turtle.goto(25, 0)
            turtle.dot(10)

turtle.hideturtle()
turtle.done()
```

Figure 4.153 Drawing the result of rolling a 6-sided die using the `turtle` module

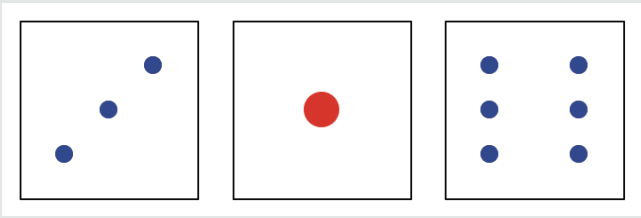


Figure 4.154 Examples of rolling a 6-sided dice using the `turtle` module

Note how the code uses `if-else` statements to decide on which dots to draw.

4.14.4 while Loops

Using an iteration construct, the algorithm in Figure 4.155 will keep asking for a new value of name as long as name remains blank.

We can achieve this in Python using the `while` statement:

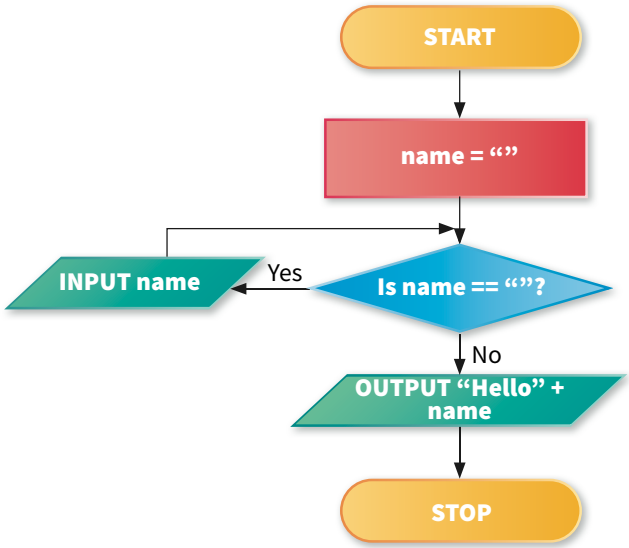


Figure 4.155 Personal greeting flowchart

```
1 name = ""
2 while name == "":
3     name = input("Enter name: ")
4 print("Hello " + name)
```

```
Enter name: Nadiah
Hello Nadiah
```

Figure 4.156 Personal greeting program

Once name is correctly entered, the program proceeds to run line 4 and name is printed with the greeting “Hello”. Since the flowchart for repeated commands will always have a loop, the `while` statement is also called a `while` **loop**.

Like the `if-elif-else` statement, Python uses indentation to determine which lines of code belong to the `while` statement. In the case of `greeting.ipynb`, only line 3 belongs to the `while` statement and it is run repeatedly as long as nothing is entered into name (in other words, as long as `name == ""` is `True`).

Note that, exactly like the flowchart, Python checks the condition after the `while` keyword at least once while the program is running and every time after the commands in the loop are complete. This means that if we set name to anything other than an empty string, Python will skip the contents of the loop.

KEY TERMS

Loop
Instructions that are repeated until a condition is met

```
# Program 4.11: greeting_skiploop.ipynb

name = "Computing"
while name == "":
    name = input("Enter name: ")
print("Hello " + name)
```

```
Hello Computing
```

Figure 4.157 Demonstration that `while` condition is always checked at least once

Additional lines with the same indentation belong to the same `while` loop. For instance, the flowchart and program in Figure 4.158 and Figure 4.159 go through the contents of a `list` and output each item on a separate line.

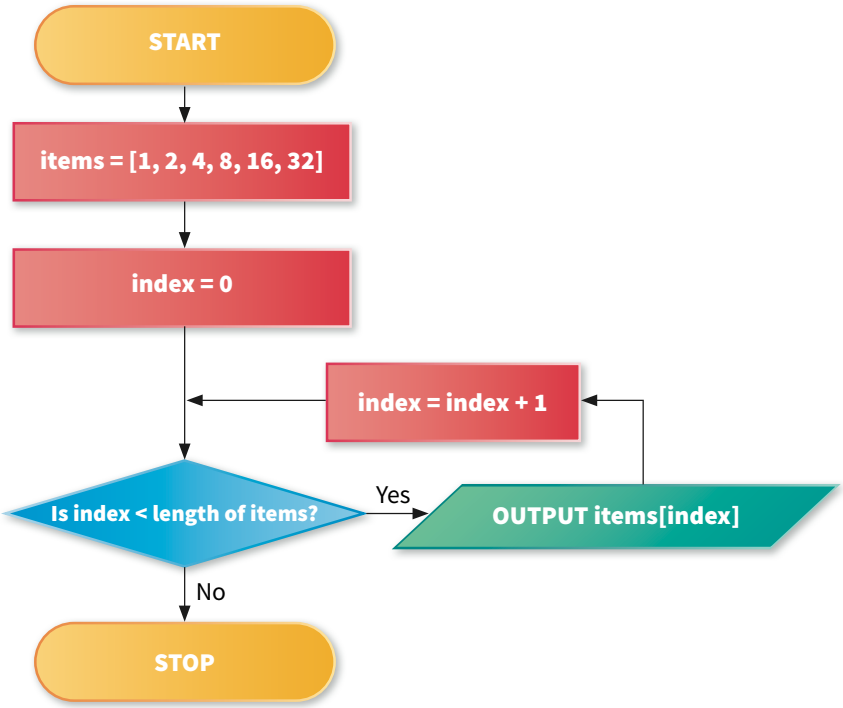


Figure 4.158 Flowchart for printing a `list`

```
# Program 4.12: printlist_while.ipynb

items = [1, 2, 4, 8, 16, 32]
index = 0
while index < len(items):
    print(items[index])
    index = index + 1
```

Figure 4.159 Program for printing a `list`

The syntax for `while` statements is summarised in Figure 4.120.

Syntax 4.13 while Statement

```
while condition:
  commands to repeat while condition is True
```

Figure 4.160 Syntax for while statement

Note that the `while` loop follows the iteration construct that we learnt previously. However, sometimes we want a convenient way for the program to go back to the start of the loop or to exit the loop early. For such cases, Python provides the `continue` and `break` keywords.

The `continue` keyword causes Python to skip the remaining commands in the loop and go straight to deciding whether the loop should run again by testing the condition after the `while` keyword. For example, suppose we have a loop that (by default) outputs "C is for word" for every word in a `list`. However, if a word does not start with C, we can decide that the default behaviour does not apply and that the program should advance to the next word instead.

The flowchart and source code in Figure 4.161 and Figure 4.162 demonstrate how such an algorithm can be implemented using the `continue` keyword:

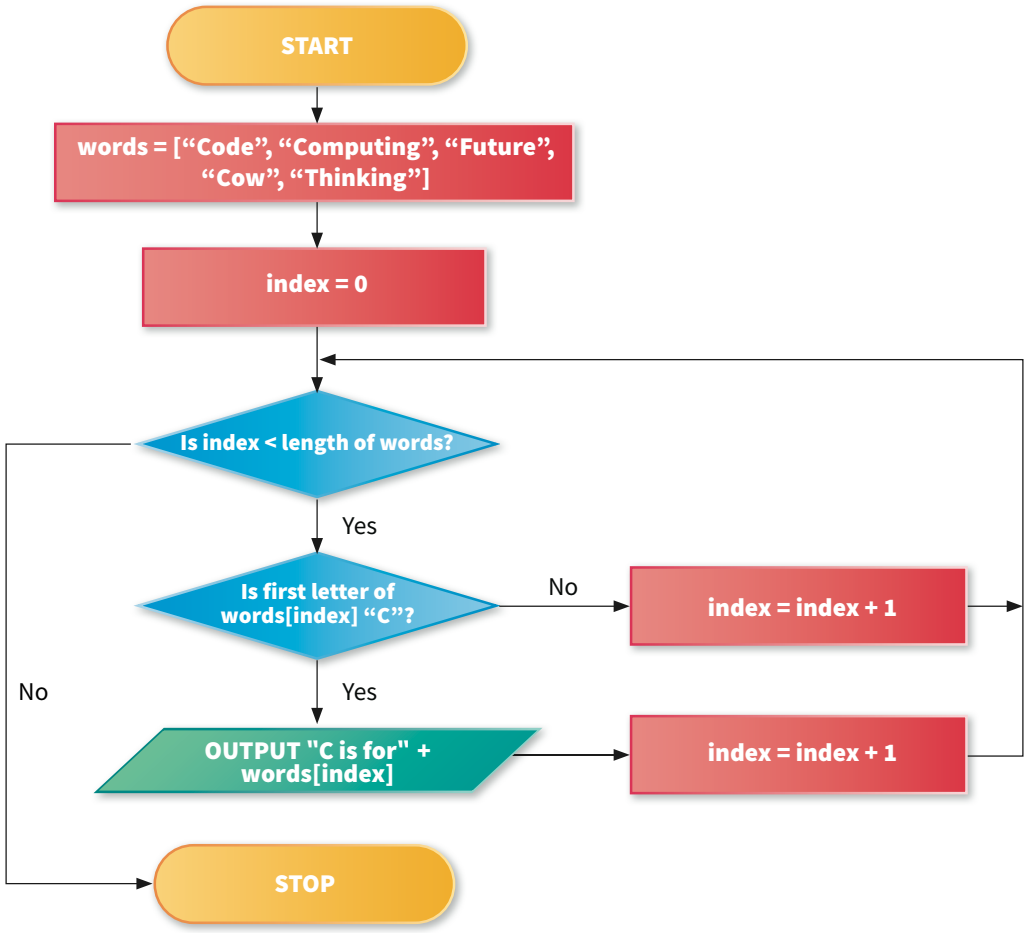


Figure 4.161 Flowchart demonstrating `continue`

```
# Program 4.13: firstletter_continue.ipynb

words = ["Code", "Computing", "Future", "Cow", "Thinking"]
index = 0
while index < len(words):
    if words[index][0] != 'C':
        index = index + 1
        continue
    print("C is for " + words[index])
    index = index + 1
```

```
C is for Code
C is for Computing
C is for Cow
```

Figure 4.162 Demonstration of using `continue` keyword to skip iterations

The `break` keyword, on the other hand, causes Python to immediately skip the remaining commands and exit the loop completely. For example, suppose we once again have a loop that (by default) outputs "C is for word" for every word in a `list`. However, this time we want the loop to end once a word that does not start with C is detected.

The flowchart and source code in Figure 4.163 and Figure 4.164 demonstrate how such an algorithm can be implemented using the `break` keyword:

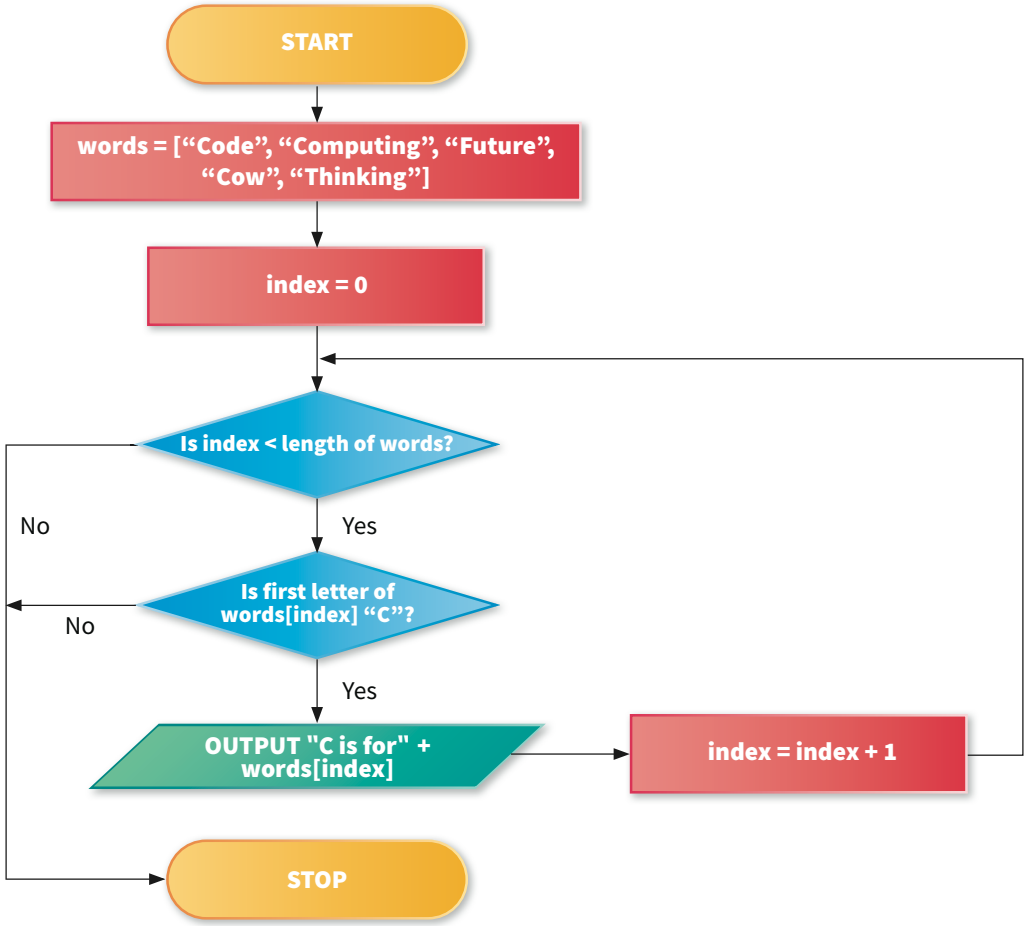


Figure 4.163 Flowchart demonstrating `break`

```
# Program 4.14: firstletter_break.ipynb

words = ["Code", "Computing", "Future", "Cow", "Thinking"]
index = 0
while index < len(words):
    if words[index][0] != 'C':
        break
    print("C is for " + words[index])
    index = index + 1
```

```
C is for Code
C is for Computing
```

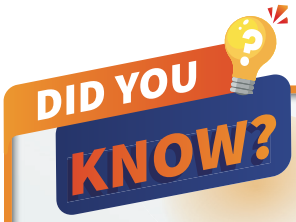
Figure 4.164 Demonstration of using `break` keyword to end a loop early

The program in Figure 4.125 demonstrates how the `continue` and `break` keywords work by playing a simple "Simon Says" game.

```
    # Program 4.14: firstletter_break.ipynb

1   # Constants
2   MAGIC_WORDS = "Simon says "
3   QUIT_COMMAND = "Quit"
4
5   print("Enter \"" + QUIT_COMMAND + "\" to end game.")
6   while True:
7       command = input("Enter command: ")
8       if command == QUIT_COMMAND:
9           break
10      elif command[:len(MAGIC_WORDS)] != MAGIC_WORDS:
11          print("We ignore the command")
12          continue
13      print("We " + command[len(MAGIC_WORDS):])
14  print("Thanks for playing!")
```

Figure 4.165 Program for playing Simon Says

In this game, the computer will repeat what the user enters as long as the input starts with the words "Simon says". This is the default behaviour that is repeated using a `while` loop that seems to repeat forever as its condition is always `True` (line 6). However, if the input matches the special word "Quit", the loop ends immediately using the `break` keyword and the game ends with the thank-you message. On the other hand, if the input does not start with "Simon says", the default behaviour is skipped and we proceed to ask for the next input instead. Try playing the game a few times and see if the new keywords behave the way you expect them to.



Run the following program that uses the `turtle` module to draw star patterns given a line segment length and turn angle:

The following examples were generated using these settings:
length = 1, angle= 1
length = 100, angle= 100
length = 100, angle= 125
length = 100, angle = 170

```
import turtle

length = turtle.numinput('Star', 'Enter segment length in pixels:')
angle = turtle.numinput('Star', 'Enter turn angle in degrees:')

turtle.color('red', 'yellow')
turtle.begin_fill()
# Loop forever
while True:
    turtle.forward(length)
    turtle.left(angle)
    # Check if turtle is close to starting point
    if abs(turtle.position()) < 1:
        # If so, break out of loop
        break
turtle.end_fill()
turtle.done()
```

Figure 4.166 Drawing star patterns using the `turtle` module

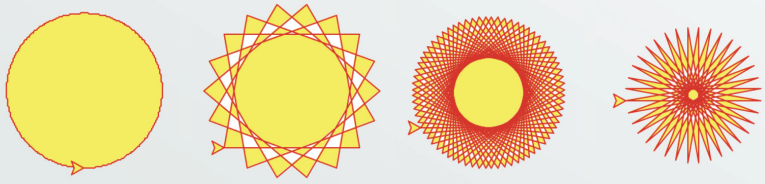


Figure 4.167 Star patterns drawn using the `turtle` module

Note how the code uses a `while` loop to keep repeating the `forward()` and `left()` method calls until it breaks out of the loop upon returning to the starting point.

4.14.5 for-in Loops

Previously, we saw an example of iterating through the items of a `list` by using an index variable:

```
# Program 4.16: printlist_while.ipynb

items = [1, 2, 4, 8, 16, 32]
index = 0
while index < len(items):
    print(items[index])
    index = index + 1
```

Figure 4.168 Iterating through a `list` using a counter

Iterating through the items of a sequence is such a common task that Python provides a shortcut for doing so: the `for-in` statement. When we use the `for-in` statement, each item in the sequence is assigned to a variable automatically before the loop is run. For instance, we can simplify the code shown in Figure 4.168 to become the code in Figure 4.169:

```
# Program 4.17: printlist_for.ipynb

items = [1, 2, 4, 8, 16, 32]
for item in items:
    print(item)
```

Figure 4.169 Iterating through a `list` without a counter using a `for-in` loop

Note that this new code no longer has an `int` counter (`index`). Instead it has a variable `item` that is assigned a new item from the `list` each time the loop repeats. The syntax for `for-in` statements is shown in Figure 4.170:

Syntax 4.14 for-in Statement

```
for name in sequence:
  commands to repeat for each item in the sequence
```

Figure 4.170 Syntax for `for-in` statement

The `for-in` loop also works with the `range()` function that you learnt earlier. As with `lists`, the loop is run for each item in the range. By using the length of a `list` as the argument for `range()`, we have an efficient way to iterate through a `list` with a counter variable.

Notice that there is no longer a need to manually increase `index` by `1` as the `for-in` loop will automatically update `index` with the next item in the provided sequence.

```
# Program 4.18: printlist_range.ipynb

items = [1, 2, 4, 8, 16, 32]
for index in range(len(items)):
    print(items[index])
```

Figure 4.171 Iterating through a `list` with a counter using a `for-in` loop and `range()`

The `for-in` loop works with `dicts` as well. The loop is run for each key-value pair in the `dict` and the loop variable is assigned to the key for each pair, as shown in Figure 4.175:

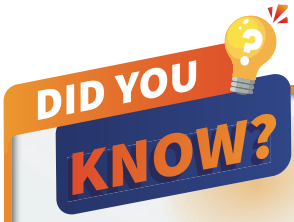
```
# Program 4.19: keys.ipynb

scores = {'Alex': 176, 'Bala': 180, 'Siti': 177}
for name in scores:
    print(name)
```

```
Alex
Bala
Siti
```

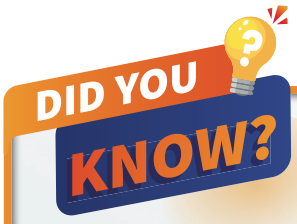
Figure 4.175 Iterating through the keys of a `dict`

Besides `lists`, `ranges` and `dicts`, the `for-in` loop can also be used with file objects to iterate through the lines of a file. This is covered later in section 4.16.

The `continue` and `break` keywords work with `for-in` loops as well.



It is not recommended to modify a Python `list` while iterating through it using a `for-in` loop. Modifying a `list` during iteration can lead to unexpected behaviour and errors. This is because internally the `for-in` loop still keeps track of a hidden counter variable and modifying the `list` during the loop can mess up this tracking.



We can shorten repetitive code using `for-in` loops. For instance, the code for drawing randomised artwork using the `turtle` module in section 4.10.7 can be shortened to the following with no change in behaviour:

```
import random
import turtle

colors = ['blue', 'green', 'red', 'orange']
for color in colors:
    turtle.color(color)
    turtle.begin_fill()
    length = random.randint(4, 10) * 10
    for i in range(3):
        turtle.forward(length)
        turtle.right(90)
    turtle.forward(length)
    turtle.end_fill()
turtle.done()
```

Figure 4.172 Shortened program for drawing randomised artwork using the `turtle` module

`for-in` loops are also used for the following program that uses the `turtle` module to draw a column chart:

```
import turtle

values = [8, 12, 2, 16, 4]
for value in values:
    for i in range(2):
        for length in [50, value * 10]:
            turtle.forward(length)
            turtle.left(90)
    turtle.forward(50)
turtle.done()
```

Figure 4.173 Drawing a column chart using the `turtle` module

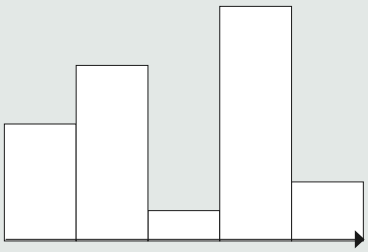


Figure 4.174 Example of column chart drawn using the turtle module

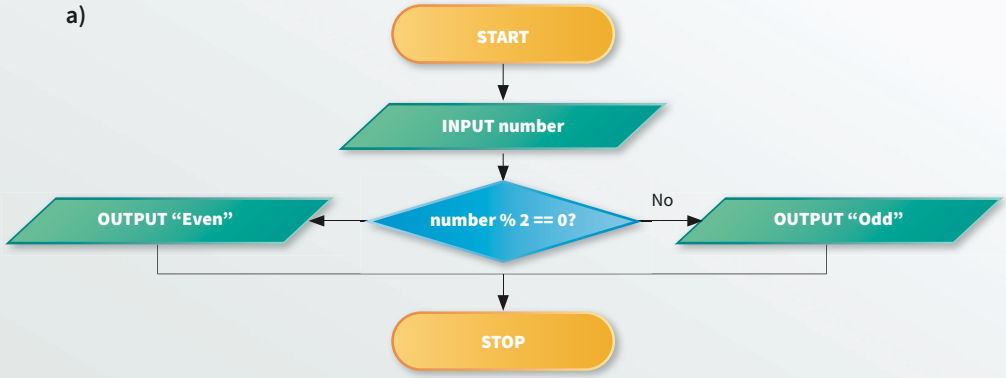
As a challenge, modify the program so each column is filled with a different colour!

QUICK CHECK 4.14

1. For each of the following flowcharts:

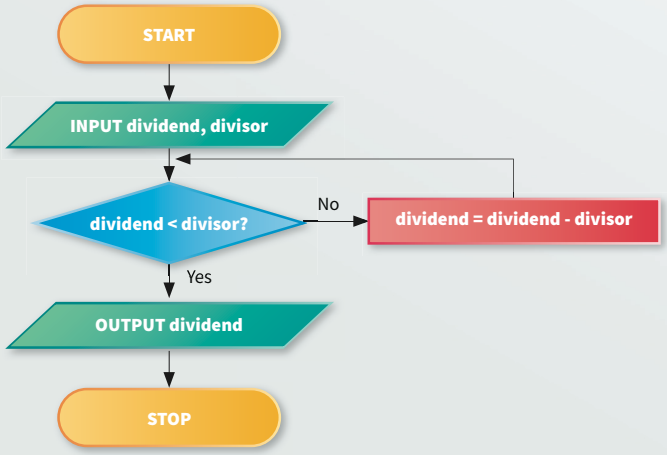
i Name the type of construct that is being used.
ii Translate the flowchart to a Python program.

a)



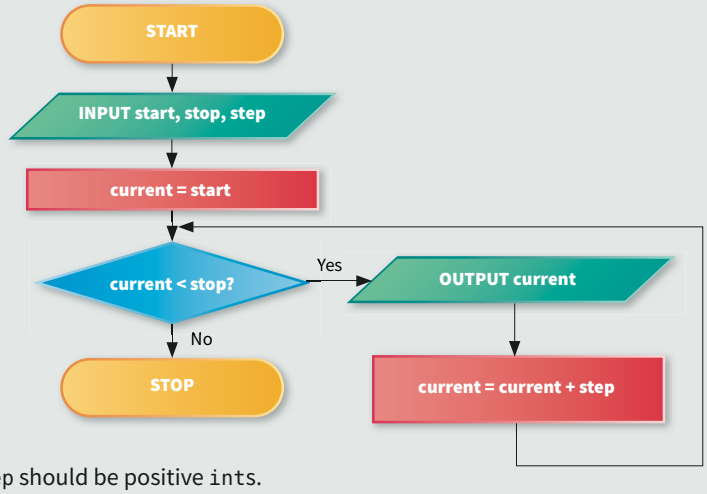
Note: `number` should be an `int`.

B)



Note: `dividend` and `divisor` should be positive `ints`.

C)



Note: `start`, `stop` and `step` should be positive `ints`.

QUICK CHECK 4.14

2. Write a program that lets the user input a `list` of `str`s by asking for each item of the `list` in order. Receiving an empty `str` from the user would indicate the end of the `list`. The program would then output the `list` on the screen.

Input	Output
• A sequence of non-empty `str`s, in order, ending with an empty `str`	• Printed `list` of all input `str`s arranged in the correct order

Table 4.49 Input and output requirements for the `list` input problem

3. Write a program that lets the user input a password. To make sure that the password has been entered correctly, the program should ask for it twice. If the two entered passwords are different, the program should output "Invalid" and repeat the input process all over again. The program should output "Valid" and end when the two entered passwords match.

Input	Output
• `password`: password entered by user • `password_again`: password entered again by user	• "Invalid" whenever `password` and `password_again` do not match, repeated as many times as needed; "Valid" when the two inputs match

Table 4.50 Input and output requirements for the password input problem

4.15 User-Defined Functions

LEARNING OUTCOMES

2.3.13 *Write and call user-defined functions that may accept parameters and/or provide a return value.*

2.3.14 *Distinguish between the purpose and use of local and global variables in a program that makes use of functions.*

Besides Python's built-in functions, you can also write your own functions. These are called user-defined functions (UDFs).

To define a UDF, we first need to choose its name. The name must be a valid Python identifier that follows the same rules described previously for variable names (see section 4.5.2.1). The UDF's name should also be different from any existing variable or function names. Otherwise, the UDF will overwrite the existing value or function associated with that name.

For example, the program `define_hello.ipynb` in Figure 4.176 defines a function named `hello()` that prints out the phrase "Hello, World!". Lines 2 and 3 form the UDF's body which is run whenever the UDF is called. Like control flow statements, this indentation is used to determine the start and end of each UDF.

If we run `define_hello.ipynb`, nothing appears on the screen because `hello()` is never actually called. In order to use a UDF, you need to call it and this can only be performed after the UDF is defined.

```
   # Program 4.20: define_hello.ipynb

1  def hello():
2      # This code does not get run at all.
3      print('Hello, World!')
```

Line 1 specifies the name.

Lines 2 and 3 form the body

Figure 4.176 Defining a UDF without calling it

The program `say_hello.ipynb` in Figure 4.177 defines the same function `hello()`, then calls the `hello()` function three times. In this program, the indentation after line 1 indicates that lines 2 and 3 belong to the `hello()` UDF's body while lines 5 to 7 do not. Also, if you place lines 5 to 7 before lines 1 to 3, running the program will produce an error.

```
   # Program 4.20: define_hello.ipynb

1  def hello():
2      # Say hello
3      print('Hello, World!')
4
5  hello()
6  hello()
7  hello()
```

```
Hello, World!
Hello, World!
Hello, World!
```

Figure 4.177 Defining and calling a UDF

When you run this program, the phrase "Hello, World!" appears on the screen three times, once for each call of `hello()` from lines 5 to 7.

The first line of a UDF is a **signature** that specifies the UDF's name and its **parameters**. Parameters are special variables that arguments are assigned to when the UDF is called. In this case, `hello()` function has no parameters. For a UDF to accept arguments, its signature must specify a parameter for every argument it expects to receive.

KEY TERMS

Parameters
Special variable in a UDF that an argument is assigned to when the UDF is called

Signature (programming)
The first line of a UDF that specifies the UDF's name and its parameters

For instance, the program `quiz.ipynb` in Figure 4.178 has a UDF `get_answer()` that expects to receive two arguments using the parameters `prompt` and `reply`. This means that when the function is called (but before the function body is run), the first argument will be assigned to a new variable named `prompt` and the second argument is assigned to a new variable named `reply`.

```
   # Program 4.22: quiz.ipynb

1  def get_answer(prompt, reply):
2      answer = input(prompt)
3      print(reply)
4      return answer
5
6  answer1 = get_answer('What is your name?', 'Thanks!')
7  answer2 = get_answer('What is your age?', 'Thanks again!')
8  answer3 = get_answer('What is your hobby?', 'Thank you!')
9
10 print('Your answers were...')
11 print(answer1, answer2, answer3)
```

This value is returned by the function after the function call ends.

Figure 4.178 Defining a UDF that accepts arguments and returns a value

Line 4 of `quiz.ipynb` also shows that a UDF can provide a return value using the `return` keyword. When Python encounters a `return` instruction, the function immediately ends and the original function call is treated as the provided return value. If no `return` instruction is encountered before the function body ends, then the default return value is None and the original function call is also treated as `None`. An example of this is the `hello()` function in Figure 4.177.

From Lines 6 to 8 of `quiz.ipynb`, `get_answer()` is called three times and the return value of each call is assigned to variables `answer1`, `answer2` and `answer3` respectively.

Figure 4.179 illustrates the process of how the arguments of a function call are assigned to new variables with names given by the function's signature and how the return value is assigned to `answer1`.

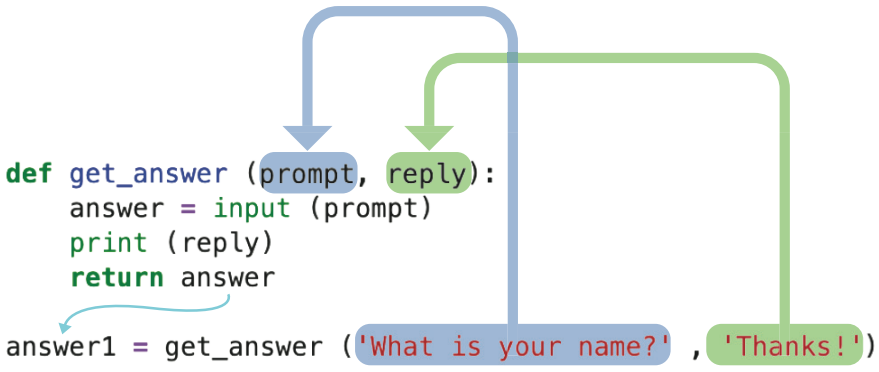


Figure 4.179 How arguments are assigned to new variables during a function call

Figure 4.180 summarises the syntax rules for a UDF that accepts either no arguments, one argument or two arguments. This pattern can be extended to write UDFs that accept even more arguments.

Syntax 4.15 User Defined Function

```
def function_name():
  commands to run when function_name is called

def function_name(parameter_1):
  commands to run when function_name is called (needs 1 argument)

def function_name(parameter_1, parameter_2):
  commands to run when function_name is called (needs 2 arguments)
```

Figure 4.180 Syntax for defining a UDF

The syntax rules for `return` are summarised in Figure 4.181. The return value after the `return` keyword is optional. If it is provided, the return value can be of any data type. If it is omitted, the return value is treated as `None`.

Syntax 4.16 return Statement

```
return

return return_value
```

Figure 4.181 Syntax for returning from a UDF

Previously, we explained that arguments are assigned to new variables based on the parameters given in the function's signature. These new variables are special because they are valid only within the function. This is also true for any variables that are created inside the function. Such variables are called **local variables** because they can only be used inside the function's body.

KEY TERMS

Local variable
A variable that is created inside a UDF and can only be used in the UDF's body

DID YOU KNOW?

Run the following program that uses the `turtle` module and allows the user to draw lines by clicking in a window:

```
import turtle

def draw_line(x, y):
    turtle.goto(x, y)

turtle.onscreenclick(draw_line)
turtle.done()
```

Figure 4.182 Allowing the user to draw lines by clicking with the `turtle` module

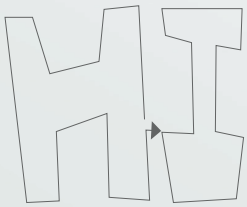


Figure 4.183 Example of drawing made by clicking with the `turtle` module

Note how the code declares a `draw_line()` function that accepts mouse coordinates as arguments. The `onscreenclick()` function then accepts `draw_line` as an argument so it is called each time the window is clicked.

The call to `onscreenclick()` also demonstrates that Python supports "first-class functions" where function names (without the brackets) are treated just like variables that can be passed as arguments to other functions.

As a challenge, you may try to shorten the code further or have the line drawn with each click be of a different colour!

For instance, the program `area.ipynb` in Figure 4.184 has a UDF named `area_of_circle()` that, when called, will assign its argument to a variable named `radius`. Inside the function, a new variable named `area` is also created:

```
   # Program 4.23: area.ipynb

1  PI = 3.14159
2
3  def area_of_circle(radius):
4      area = PI * radius ** 2
5      # Both radius and area are valid variables here
6      return area
7
8  print(area_of_circle(2))
9  # However, radius and area are NOT valid variables here
```

Figure 4.184 Defining a UDF that uses local variables

In this case, both `radius` and `area` are local variables for `area_of_circle()`. This means that outside the function body of `area_of_circle()`, `radius` and `area` are not defined. For example, commands like `print(radius)` and `print(area)` would run correctly inside the function body (e.g., line 5), but the same code would produce an error message outside the function body (e.g., line 9).

Notice that `area_of_circle()` uses the constant `PI` that is defined outside of the function body. Besides local variables, the code inside a function body also has read access to the **global variables**, constants and functions that are defined outside of the function. The different local as well as global variables and constants found in `area.ipynb` are illustrated in Figure 4.70:

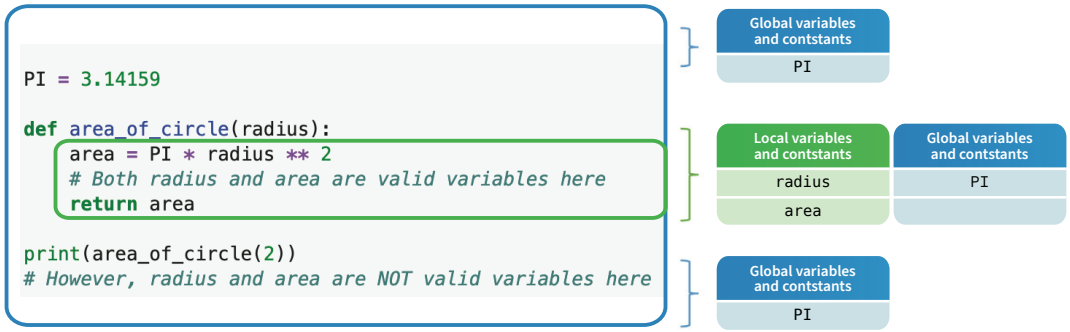


Figure 4.185 Local and global variables and constants in `area.ipynb`

In a typical program, constants and functions are defined once and do not change. Hence, it is usually predictable to use global constants and call global functions such as `input()`, `print()` or other UDFs inside a function body.

KEY TERMS

Global variable
A variable that is created outside of a UDF and is readable from the UDF's body if its name is not hidden by a local variable

Variables are different from constants and functions as they can change while a program is running. A function that depends on global variables may produce completely different behaviour each time it is called. To avoid this, UDFs should request for all the inputs that they need as arguments instead of using global variables. For instance, the program `add_function.ipynb` in Figure 4.186 shows two ways of writing a function to add two numbers. In general, the `good_example()` approach is better than the `bad_example()` approach as all the inputs that can affect the return value are clearly specified in the function's signature and the function body does not use any global variables.

```
    # Program 4.24: add_function.ipynb

1   def good_example(num1, num2):    # num1 and num2 are local
2       return num1 + num2           # OK
3
4   def bad_example():               # num1 and num2 are global
5       return num1 + num2           # Don't do this!
6
7   num1 = 19
8   num2 = 65
9   print(good_example(20, 17))
10  print(bad_example())
```

Figure 4.186 Inputs to functions should be provided as arguments instead of global variables

For `add_function.ipynb`, the variable names `num1` and `num2` are used in two different contexts:

1. On lines 1 and 2, they are local variables for the `good_example()` function.
2. On lines 5, 7 and 8, they are global variables.

While they have the same names, it is important to understand that the local variables named `num1` and `num2` on lines 1 and 2 are completely separate variables from the global variables named `num1` and `num2` on lines 5, 7 and 8.

When `good_example()` is called on line 9, two new local variables are created and given the names `num1` and `num2` just before the function body on line 2 is run. This temporarily hides the global variables that are also named `num1` and `num2` and prevents them from being accessed inside the body of `good_example()`.

However, both global variables are merely hidden and not lost. After the call to `good_example()` ends, the local versions of `num1` and `num2` are removed and the global variables that are named `num1` and `num2` become accessible again with the same values that were assigned to them previously.

Figure 4.187 illustrates the local and global variables in `add_function.ipynb` and how having a local variable can hide access to global variables with the same name.

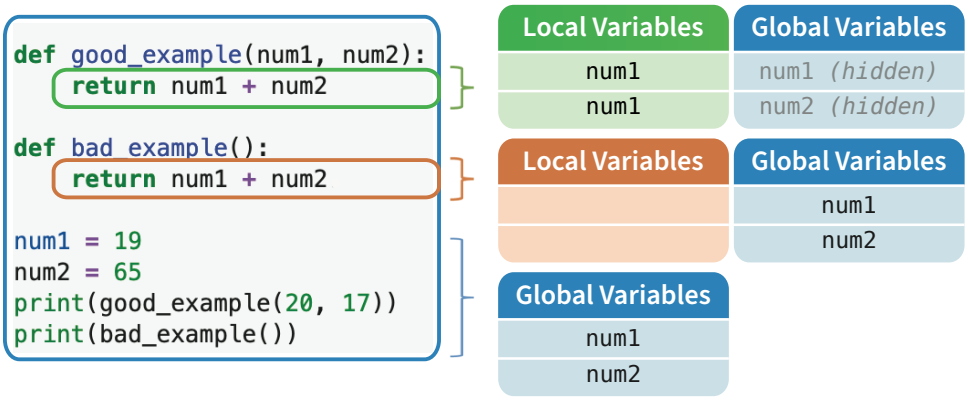
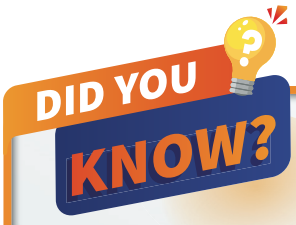


Figure 4.187 Local and global variables and constants in `add_function.ipynb`

Notice how the global variables named `num1` and `num2` are hidden for `good_example()` but are not hidden for `bad_example()`.

UDFs have several advantages:

- Code that is repeated in different locations of the program can be put in UDFs. The repeated code can then be replaced with shorter function calls.
- When designing an algorithm, the solution to each sub-problem can be placed in a separate UDF. Each UDF will be smaller than the main program and thus can be more easily understood. (See Chapter 7.)
- When working in a team, different programmers can be given responsibility for different UDFs so they can all work concurrently.
- Each UDF can be tested separately and more thoroughly with different inputs compared to a single large program with no UDFs. (See Chapter 6.)

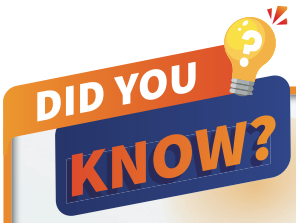


In this textbook, we will limit ourselves to read access for global variables. If you accidentally try to reassign a global variable, you may find that your existing code no longer works. For example, let us consider a simple function `example()` that prints the value of a global variable `number`:

```
# Program 4.25: print_global.ipynb

def example():
    print(number)

number = 2024
example()
```

```
2024
```

Figure 4.188 Printing a global variable



This program works fine. However, if we add a line that tries to reassign the global variable `number`, we suddenly get an error:

```
1 def example():
2     print(number)
3     number = number + 1
4
5 number = 2024
example()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[1], line 6
      3     number = number + 1
      5 number = 2024
----> 6 example()

Cell In[1], line 2, in example()
      1 def example():
----> 2     print(number)
      3     number = number + 1

UnboundLocalError: cannot access local variable 'number' where it is not associated with a
value
```

Figure 4.189 Error when attempting to modify a global variable

This may be doubly surprising since the line we added (line 3) comes *after* where the error occurs (line 2).

The explanation is that Python code in a function definition is treated differently from top-level code that is not in a function definition:

- Top-level code is run immediately as Python reads it and all variables are global.
- On the other hand, code in a function definition is not run immediately and there is a need to distinguish between local and global variables. Python does this by reading all the code in the function in advance (without running it) and identifying which variables are reassigned anywhere in the function. These variables are then treated as local variables for the entire function body.

In this case, Python identifies that the code added on line 3 will reassign `number` and thus treats `number` on line 2 as a local variable. This then leads to an error when the function is called because, as a local variable, number is not initialised yet on line 2.

DID YOU KNOW?

Python allows you to override this behaviour by declaring that `number` must be treated as a global variable by using the `global` keyword. This also grants the function write access to reassign the specified global variable:

```
# Program 4.27: print_and_modify_global_fixed.ipynb

def example():
    global number
    print(number)
    number = number + 1

number = 2024
example()

# Note: global variable number is modified after call
print(number)
```

```
2024
2025
```

Figure 4.190 Printing and modifying a global variable

Although this works, it is generally a bad practice to reassign global variables inside a function and use of the `global` keyword is not recommended.

QUICK CHECK 4.15

1. What is the output of the following code?

```
def decorate(s, symbol):
    return symbol + s + symbol

message = 'HELLO'
message = decorate(message, '*')
message = decorate(message, '!')
print(message)
```

a) `!*HELLO!*`
b) `!*HELLO*!`
c) `*!HELLO!*`
d) `*!HELLO*!`

2. Predict the output of the following programs:
a)

```
def add_one(x):
    return x + 1

x = 2024
add_one(x)
print(x)
```

QUICK CHECK 4.15

b)

```
def add_one(x):
    return x + 1

x = 2024
x = add_one(x)
print(x)
```

c)

```
def add_one(x):
    x = x + 1

x = 2024
add_one(x)
print(x)
```

d)

```
def add_one(x):
    x = x + 1

x = 2024
x = add_one(x)
print(x)
```

e)

```
def add_one():
    return x + 1

x = 2024
x = add_one()
print(x)
```

f)

```
def add_one():
    return x + 1

x = 2024
y = add_one()
y = add_one()
print(y)
```

4.16 with Statements

LEARNING OUTCOMES

2.3.4 *Use the open() built-in function as well as the read(), readline(), write() and close() methods to perform non-interactive file input/output.*

In programming, we often need to make use of the computer's limited resources. For some resources such as memory for storing variables, Python takes care of reserving and releasing memory automatically. However, there are other kinds of resources that require the programmer to manage reserving and releasing them properly. Examples of such resources include network connections and files in secondary storage.

If a resource is accidentally reserved but never released, then it becomes unavailable for other programs. This is like borrowing a book from a library without returning it or starting a phone call and never hanging up.

To help programmers manage such resources properly, Python offers `with` statements. `with` statements are designed to ensure that resources which are reserved when the statement starts are automatically released when the statement ends. Specifically, we shall use `with` statements to perform file input/output using Python.

4.16.1 File Input/Output Functions

The built-in function `open()` is used for opening files on your computer for either input and output based on its second argument. To use a file for input, use "r" (meaning "read") for the second argument. To use a file for output, use "w" (meaning "write" to overwrite the file) or "a" (meaning "append" to continue from the end of an existing file) for the second argument.

Function	Argument(s)	Description	Example
`open()`	A required `str` and an optional `str`	Opens file with name given by first argument. The second argument is either `'r'`, `'w'` or `'a'` and determines if the file is open for reading, overwriting, or appending. Opens file for reading if second argument is not given. Returns a file object (see below).	`f = open('file.txt', 'w')` `f.write('Line 1\nLine 2\n')` `f.close()` `f = open('file.txt')` `print(f.read())` `f.close()` `Line 1` `Line 2`

Table 4.51 File input and output functions

4.16.2 File Input/Output Methods

The file object that is returned by open( ) has the following methods:

File Object Methods	Argument(s)	Description	Example
`close()`	None	Closes the text file and saves any changes; not necessary if file object is used with a `with` statement Returns None.	`# Writes str to file.txt` `f = open('file.txt', 'w')` `f.write('Line 1\nLine 2\n')` `f.close()`
`read()`	None	Returns entire text file (including newline characters) as a `str`. The file object must have been open for reading.	`# Prints out file.txt` `with open('file.txt') as f:` `  contents = f.read()` `print(contents)` `Line 1` `Line 2`
`readlines()`	None	Returns next line from the text file (including a newline character at the end if present) or an empty `str` if already at end of file. The file object must have been open for reading.	`# Prints out first line` `with open('file.txt') as f:` `  line = f.readline()` `print(line)` `Line 1`
`readline()`	None	Returns a `list` of all the lines in the text file. Each line includes a newline character at the end if present. The file object must have been open for reading.	`# Prints out list of lines` `with open('file.txt') as f:` `  lines = f.readlines()` `print(lines)` `['Line 1\n', 'Line 2\n']`
`write()`	A `str`	Writes the argument into the text file. Does not automatically add a newline character. The file object must have been open for overwriting or appending. Returns None.	`# Writes str to file.txt` `with open('file.txt',` `  'w') as f:` `  f.write('Line 1\n' +` `  'Line 2\n')`

4.16.3 Opening and Closing Files

To avoid losing data, it is important to always close any files you open. If a `with` statement is not used, the `close()` method must be called as the last instruction for each file object that is returned by `open()`. However, if a `with` statement is used, the `close()` method is automatically called when the `with` statement completes.

The syntax of the `with` statement is as follows:

Syntax 4.17 with Statement

```
with expression:
    commands to run while resource is reserved

with expression as variable_name:
    commands to run while resource is reserved
```

Figure 4.191 Syntax for `with` statement

The value that appears after `with` must meet certain requirements so Python knows how to release the resource when the with statement ends. File objects returned by `open()` are one such type of value. For file input/output, we usually need access to the file object inside the statement to call its methods, so we would use the second form of the `with` statement to assign the file object a variable as well.

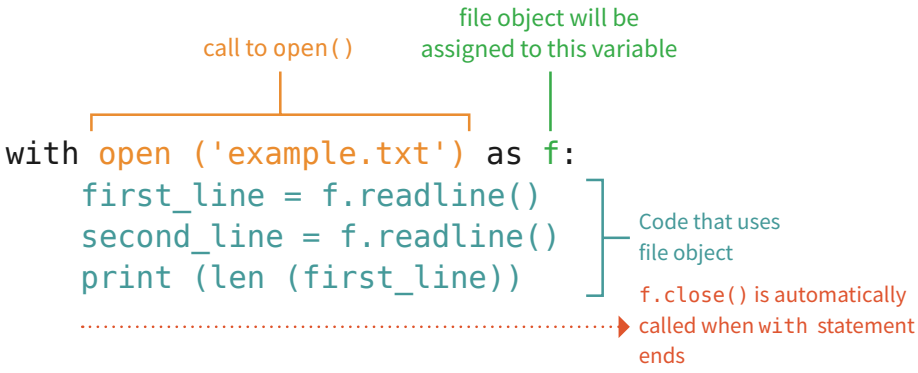


Figure 4.192 Illustration of `with` statement syntax and how `close()` method is called automatically

4.16.4 Newlines in File Input/Output

File objects can be used with `for` loops to iterate through the file's lines of text. However, how `for` loops, `readline()` and `readlines()` treat newline characters may need some explanation.

In JupyterLab Desktop, under the File menu, create a new text file with the contents shown in Figure 4.193. Take care not to end line 3 with a newline character:

Program 4.28: example.txt

```
Line 1
Line 2
Line 3
```

Figure 4.193 Example text file

If we use a `for` loop to print out the lines directly, we will obtain extra blank lines:

```
# Program 4.29: print_lines_unexpected.ipynb

# This notebook must be in the same folder as example.txt
with open('example.txt') as f:
    for line in f:
        print(line)
```

```
Line 1

Line 2

Line 3
```

Figure 4.194 Extra blank lines are output when printing the lines of a text file with a `for` loop

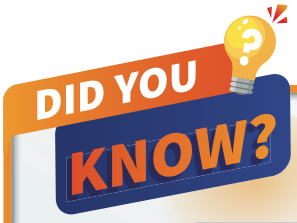
To understand why, let us call `readlines()` instead and print what is returned. We see that a `list` of three items (one for each line) is returned as expected. However, what may *not* be expected is that the `'\n'` character is included for lines 1 and 2.

```
with open('example.txt') as f:
    lines = f.readlines()
    print(lines)
```

```
['Line 1\n', 'Line 2\n', 'Line 3']
```

Figure 4.195 Inspecting the `list` returned by `readlines()`

This is different from what we are used to with `input()` and `print()`, which handle newlines automatically. For `input()`, when the user presses enter to submit, the newline that is produced by pressing enter is automatically excluded from the return value. For `print()`, a newline is automatically included at the end of each call so the next output starts on a separate line.



To remove or replace the automatic newline at the end of each `print()` call, use the end keyword argument:

```
print('Hello, ')           # print() normally ends with a newline
print('world!')
```

```
Hello,
world!
```

```
print('Hello, ', end='')    # Removes newline
print('world!')
```

```
Hello, world!
```

```
print('Hello, ', end='!')   # Replaces newline
print('world!')
```

```
Hello, !world!
```

Figure 4.196 The `print()` function accepts an end keyword argument that is appended to the output

On the other hand, for file input/output, newlines must be handled manually. In particular, `for` loops, `readline()` and `readlines()` will not automatically remove trailing newlines. Combined with the default behaviour of `print()`, which automatically includes a newline at the end of each call, this explains why there are extra blank lines in the output for Figure 4.194.

One possible solution is to test each line manually to see if it ends with `'\n'`. Figure 4.197 shows three different alternatives that use this method to print the contents of `example.txt` correctly without extra blank lines. The three alternatives use a `for` loop, `readline()` and `readlines()` respectively:

```
with open('example.txt') as f:
    for line in f:
        if line.endswith('\n'):
            line = line[:-1]
        print(line)
```

```
Line 1
Line 2
Line 3
```

```
with open('example.txt') as f:
    line = f.readline()
    while line != '':
        if line.endswith('\n'):
            line = line[:-1]
        print(line)
        line = f.readline()
```

```
Line 1
Line 2
Line 3
```

```
with open('example.txt') as f:
    lines = f.readlines()
    for line in lines:
        if line.endswith('\n'):
            line = line[:-1]
        print(line)
```

```
Line 1
Line 2
Line 3
```

Figure 4.197 Three alternatives for manually testing if each line ends with a newline

Note that an `if` statement with `endswith()` is needed before slicing off the last character because the last line of the file may not end with `'\n'`. Without this, we may accidentally slice off the last character of the file (note the missing "3"):

```
with open('example.txt') as f:
    for line in f:
        line = line[:-1]
        print(line)
```

```
Line 1
Line 2
Line
```

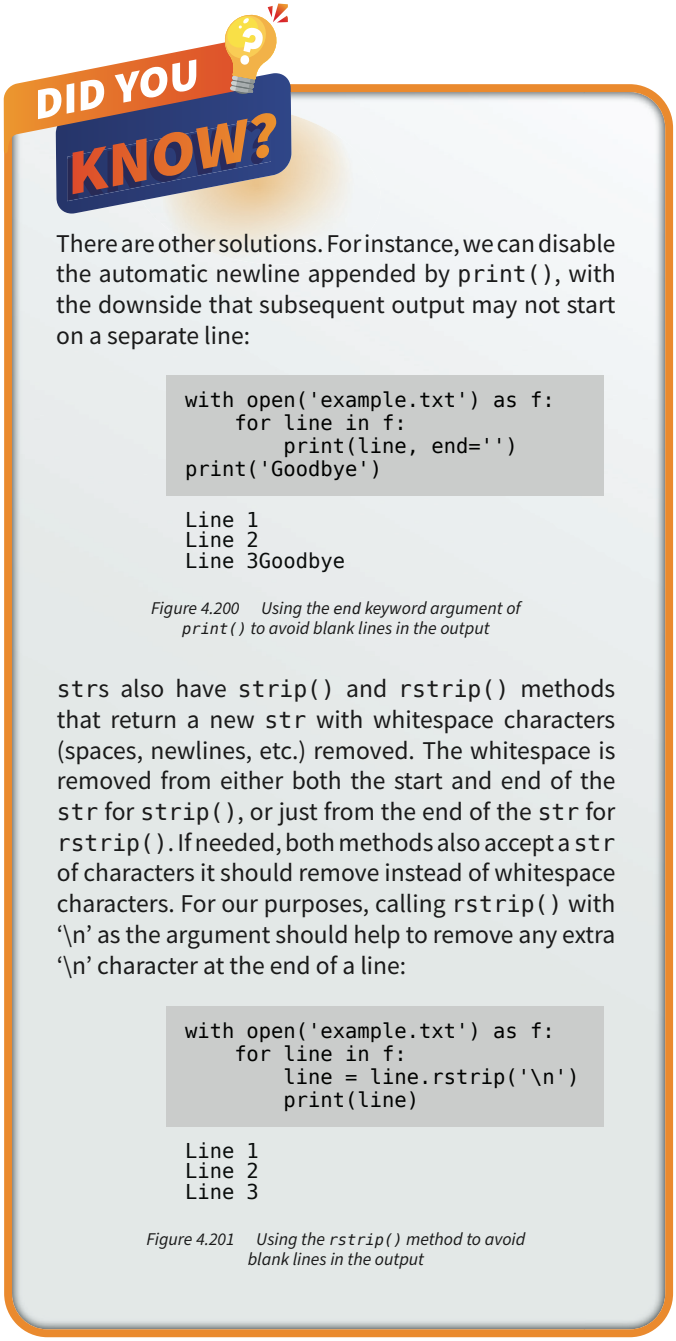
Figure 4.198 Incorrect example of slicing off the last character of each line without testing

Alternatively, we can call `read()` to read the entire file as a `str` and use the `split()` method with `'\n'` as the delimiter. Compared to `readlines()`, this technique has the advantage of consistently removing `'\n'` characters from all the extracted lines:

```
with open('example.txt') as f:
    lines = f.read().split('\n')
    for line in lines:
        print(line)
```

```
Line 1
Line 2
Line 3
```

Figure 4.199 Using `split()` with a newline delimiter



There are other solutions. For instance, we can disable the automatic newline appended by `print()`, with the downside that subsequent output may not start on a separate line:

```
with open('example.txt') as f:
    for line in f:
        print(line, end='')
print('Goodbye')
```

```
Line 1
Line 2
Line 3Goodbye
```

Figure 4.200 Using the end keyword argument of `print()` to avoid blank lines in the output

`strs` also have `strip()` and `rstrip()` methods that return a new `str` with whitespace characters (spaces, newlines, etc.) removed. The whitespace is removed from either both the start and end of the `str` for `strip()`, or just from the end of the `str` for `rstrip()`. If needed, both methods also accept a `str` of characters it should remove instead of whitespace characters. For our purposes, calling `rstrip()` with '\n' as the argument should help to remove any extra '\n' character at the end of a line:

```
with open('example.txt') as f:
    for line in f:
        line = line.rstrip('\n')
        print(line)
```

```
Line 1
Line 2
Line 3
```

Figure 4.201 Using the `rstrip()` method to avoid blank lines in the output

Similarly, for the `write()` method, we must explicitly include newline characters after each line of output. There is no option like for `print()` to automatically include a newline character at the end of each call to `write()`.

QUICK CHECK 4.16

1. Convert the following program to use a `with` statement instead of calling `close()`:

```
f = open('cat.txt', 'w')
f.write(' /v\ \n')
f.write('(o.o)\n')
f.write(' >^< \n')
f.close()
```

2. Create the following input file:

#	**Program 4.28: `run_count.txt`**
1	0

Figure 4.202 Input file for the run count program

Write a program that performs the following each time it is run:

- Reads the value in `run_count.txt` as an `int`
- Increases the value by 1
- Prints the value on the screen
- Overwrites the value in `run_count.txt`

REVIEW QUESTION

1. Alex wishes to write a program to find out which loaned books are overdue in the school library.

a) Alex studies the problem and decides to use `str` values in the format `'YYYY-MM-DD'` to represent day `DD`, month `MM` and year `YYYY`. Table 4.53 shows some examples of how dates would be represented in Alex's program.

Date	**`str` value**
1 January 2025	`2025-01-01`
22 October 2025	`2025-10-22`
11 November 2025	`2025-11-11`
1 January 2026	`2026-01-01`

Table 4.53 Representation of dates in Alex's program

Explain why Alex's method of representation will simplify his program later when he needs to compare dates.

b) The input and output requirements for Alex's problem are provided in Table 4.54. Write a program that processes the provided inputs and correctly prints out the titles of all the overdue books. In your program, write and use a UDF that takes in the due date of a book and today's date as input arguments, then returns `True` if the due date has passed and `False` otherwise. Assume that the input data will always be valid.

REVIEW QUESTION

Input	Output
• `today`: `str` representation of today's date • `titles`: titles of loaned books (via text file) • `duedates`: corresponding `str` representation of due dates (via text file)	• Titles of all loaned books with due dates earlier than `today`

Table 4.54 Input and output requirements for the problem of overdue books using lists

Some input files and a program template are provided as follows:

#	**Program 4.32: `titles.txt`**
1	How to Solve a Mystery
2	Vacant Memories
3	The Cybersnake Chronicles
4	Music History
5	Like Tears in Rain
6	Out of the Abyss

Figure 4.203 Input file containing titles of loaned books

#	**Program 4.33: `duedates.txt`**
1	2024-07-27
2	2024-07-04
3	2024-07-11
4	2024-06-30
5	2024-07-03
6	2024-07-07

Figure 4.204 Input file containing corresponding str representation of due dates

#	**Program 4.34: `overdue_template.ipynb`**
1	`# This program outputs the titles of all overdue books.`
2	
3	`# Input`
4	`today = input("Enter today's date in YYYY-MM-DD format: ")`
5	`with open('titles.txt') as f:`
6	`    titles = f.read().split('\n')`
7	`with open('duedates.txt') as f:`
8	`    duedates = f.read().split('\n')`
9	
10	`# Process and Output`
11	`...Fill in this section...`

Figure 4.205 Program template

2. Write a program to extract the hour, minute and second values (as `int`s) from a time string that is provided in the format "HH:MM:SS". Assume that the input data will always be valid.

REVIEW QUESTION

Input	Output
• `time_str`: time string in the format "HH:MM:SS", where the hour value must be from 0 to 23 inclusive, the minute value must be from 0 to 59 inclusive, and the second value must be from 0 to 59 inclusive	• Hour value, minute value and second value, each on a separate line in the above order

Table 4.55 Input and output requirements for the problem of reading a time string

3. Write a program to calculate the number of seconds between a start time and an end time, which are both provided in the format "HH:MM:SS". In your program, write and use a UDF that takes in a time string as an input argument, then returns a `list` containing the time string's hour value, minute value and second value in that order. Assume that the input data will always be valid, the end time is after the start time, and both times occur on the same day.

Table 4.56 Input and output requirements for the time interval problem

Input	Output
• `start_time`: start time in the format "HH:MM:SS", where the hour value must be from 0 to 23 inclusive, the minute value must be from 0 to 59 inclusive, and the second value must be from 0 to 59 inclusive • `end_time`: end time in the format "HH:MM:SS", where the hour value must be from 0 to 23 inclusive, the minute value must be from 0 to 59 inclusive, and the second value must be from 0 to 59 inclusive	• Number of seconds between the start time and end time

(Hint: HH:MM:SS is HH × 60 × 60 + MM × 60 + SS seconds after midnight.)