Download Chapter pdf

Download Chapter notebook (ipynb)

Mandatory Lesson Feedback Survey

- What are I/O operations?
- What do variables do?
- Why types and scopes of variables are important?
- What types of operations are used?

- Understanding the output and input operations
- Build concepts of different types of variables
- Learning about type conversions and scope
- Understanding mathematical and logical operations

In programming, we process data and produce outputs. When data is being processed, it is stored in a memory, so that it is readily available, and can therefore be subject to the processes we want to apply.

Throughout this section, we will discuss how to handle data in Python. We start by displaying data on the screen, and see how to receive input from a user. We then use these techniques to perform different mathematical and logical operations. This chapter introduces the fundamental principles that we employ every time we code in Python. On that account, make sure you understand everything before moving on.

I/O Operations

In computer science, input or output operations refer to the communication between an information processing system such as a computer, and the outside world, which may be a user or another computer. Such communications are more commonly known as *I/O operations*. In general, the outside world — especially in the context of this course, may be loosely defined as anything that falls outside of the coding environment.

REMEMBER

Only what we define within the environment and what we store in the memory is directly controlled by our application. We may access or take control over other environments through certain mediums; however, such interactions are classified as I/O operations. An example of this is interacting with a file on our computer, which we discuss in the topic of Strings. Whilst we have complete control over a file while working on it (e.g. reading from it or writing to it), the access to the file and the transmission of

data is in fact controlled and managed not by the programming environment but by the operating system of the computer.

In programming, I/O operations include, but are not limited to:

- displaying the results of a calculation
- require the user to enter a value
- writing or reading data to and from a file or a database
- downloading data from the Internet
- operating a hardware (e.g. a robot)

Advanced Topic

If you are interested in learning more about I/O systems and how they are handled at operating system level, you might benefit from chapter 13 of *Operating Systems Concepts, 8th ed.* by Abraham Silberschatz, Greg Gagne, and Peter Galvin.

I/O Operations in Python

Input and Output

In this section, we learn about two fundamental methods of I/O operations in Python. We will be using these methods throughout the course, so it is essential that you feel comfortable with them and the way they work before moving on.

Producing an output

Print

The term `output` in reference to an application typically refers to data that has either been generated or manipulated by that application.

For example; we have two number and we would like to calculate their sum. The action of calculating the sum is itself a *mathematical operation* (discussed in the coming section). The result of our calculation is called an `output`. Once we obtain the result, we might want to save it in a file or display it on the screen, in which case we will be performing an *I/O operation*.

The simplest and most frequently used method for generating an output in almost every modern programming language is to display something on the screen. We recommend using JupyterLab

notebooks to run our scripts and the typical method to produce an output is to display it in cell below the code.To do this, we will call a dedicated built-in function named print().

REMEMBER

In programming, a `function` is essentially an isolated piece of code. It usually to takes some inputs, *does* something to or with them, and produces an `output`. The pair of (typically round) parenthesis that follow a function are there to provide the function with the *input arguments* it needs when we *call* it, so that it can do what it is supposed to do using our data. We will explore functions in more details in Lesson 4 Functions.

The print() function can take several inputs and performs different tasks. Its primary objective, however, is to take some values as input and display them on the screen. Here is how it works:

Suppose we want to display some text in the terminal. To do so, we write:

```
1 print('Hello world!')
```

in a cell of our notebook (or, if not using a notebook, an editor or IDE) and save the notebook in a file. This is now a fully functioning Python programme that we can run using the Python interpreter.

If you are using an Integrated Development Environment (IDE) — e.g. Visual Studio Code, you have to save the code in a file with extension `.py` and may then execute your code using the internal tools provided by that IDE. The specifics of how you do so depend on the IDE that you are using.

`.py` Python scripts can also be executed manually. To do so, we open the terminal in MacOS or Linux or the command prompt (CMD) in Windows and navigate to the directory where we saved the script.

NOTE

If you don't know how to navigate in the terminal, see the example in section How to use terminal environment? at the end of this chapter.

Once in the correct directory, we run a script called `script_a.py` by typing `python3 script_a.py` in our terminal as follows:

```
1 python3 script_a.py
```

```
1 Hello world!
```

This will call the Python 3 interpreter to execute the code we wrote in `script_a.py`. Once executed, which in this case should be instantaneously, we should see the output.

In a JupyterLab notebook we can press the keyboard shortcut 'shift-enter' to execute the code in a cell. The output will be displayed below the code cell.

Congratulations you have now successfully written and executed your first programme in Python.

REMEMBER

We know print() is a *function* because it ends with a pair of parenthesis, and it is written entirely in lowercase characters PEP-8: Function Names. Some IDEs change color when they encounter built-in functions in the code so that we won't accidentally overwrite them. We shall discuss *functions* in more details in Lesson 4 Functions.

We can pass more than a single value to the print() function, provided that they are separated with a comma. For instance, if we write the code below and run the script, the results would be as shown in *output*.

```
print('Hello', 'John')
```

```
Hello John
```

Notice that there is a space between 'Hello' and 'John' even though we did not include a space in our text. This is the default behaviour of the print() function when it receives more than a single value (argument).

This default behaviour may be changed:

```
print('Hello', 'John', sep='')
```

```
HelloJohn
```

```
print('Hello', 'John', sep='--')
```

```
Hello--John
```

```
print('Jane', 21, 'London', sep='.')
```

```
Jane.21.London
```

Do it Yourself

Write code that displays the following output:

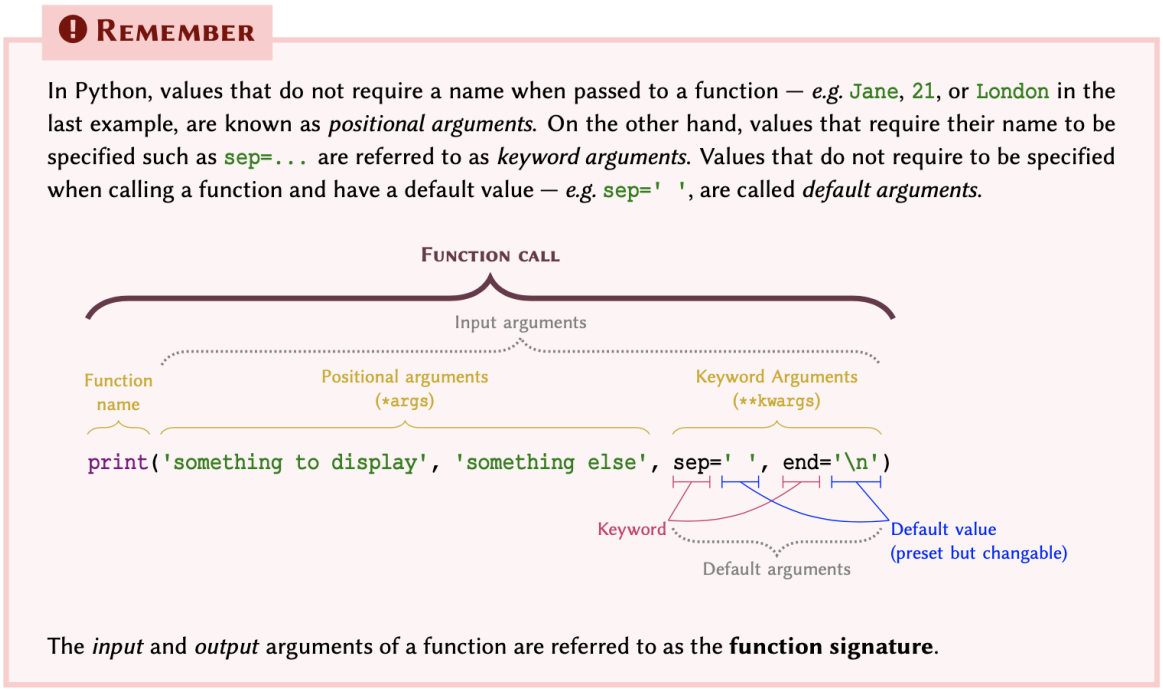


Figure 1: Explanation of a function call

> **Protein Kinase C (Alpha subunit)**

```
print('Protein Kinase C (Alpha subunit)')
```

```
Protein Kinase C (Alpha subunit)
```

Receiving an input

Input

Inputs are I/O operations that involve receiving some data from the outside world. This might include reading the contents of a file, downloading something from the Internet, or asking the user to enter a value.

The simplest way to acquire an input is to ask the user to enter a value in the terminal. To do so, we use a dedicated built-in function called input().

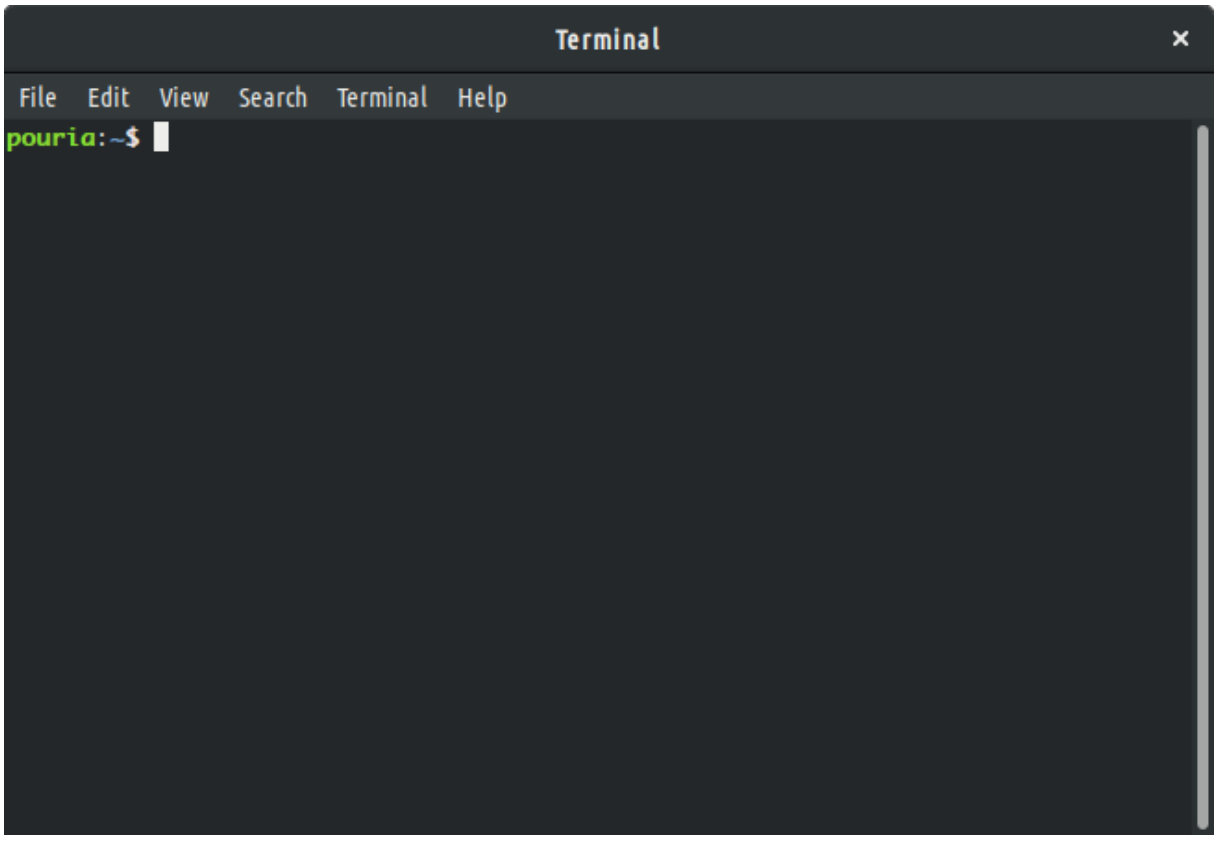


Figure 2: Terminal window on a Linux computer

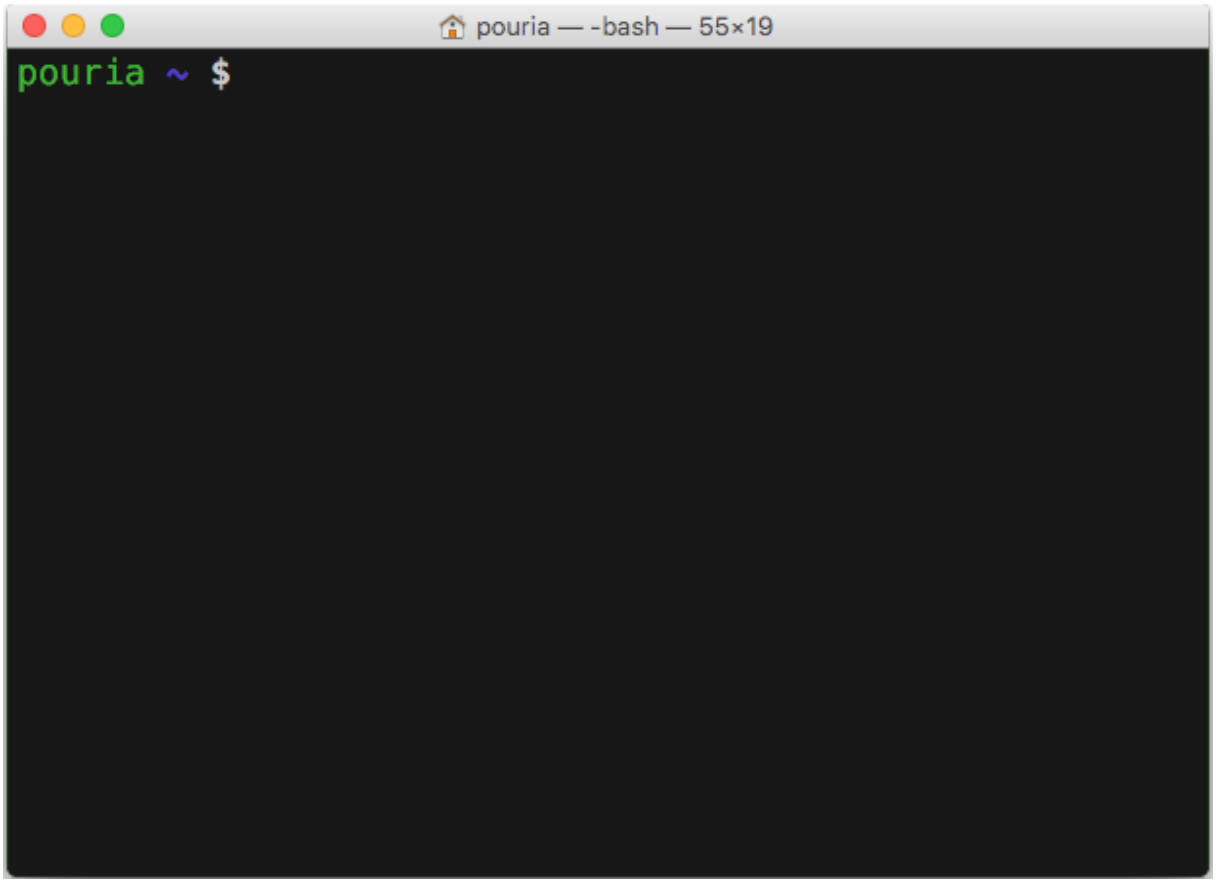


Figure 3: Terminal window on a Mac

Note

In a Unix system (Mac OS or Linux), a tilde (~) is an alias for a user's home directory.

The function takes a single *argument* called `prompt`. Prompt is the text displayed in the terminal to ask the user for an input. Figure Terminal window on a Linux computer and Terminal window on a Mac, illustrates a screen shot of my personal computer's prompt, where it displays my user name (i.e. `pouria`) followed by a tilde (~). A terminal prompt may be different in each computer and operating system.

Here is how we implement the input() function:

```
1 input('Please enter your name: ')
```

which is exactly the same as:

```
1 input(prompt='Please enter your name: ')
```

If we save one of the above in a notebook and execute it, we will see:

```
1 python3 script_b.py
2
3 Please enter your name: _
```

The terminal cursor, displayed as an underscore in our example, will be in front of the prompt (i.e. `'Please enter your name: '`) waiting for a response. Once it receives a response, it will proceed to run the rest of the code (if any), or terminate the execution.

We may store the user's response in a variable. Variables are the topic of the next section, where we shall also review more examples on input() and how we can use it to produce results based on the responses we receive from the user.

Remember

Python is an interpreted language; that is, the code we write is executed by the Python interpreter one line at a time. The input() function performs a *blocking* process. This means that the execution of the code by the Python interpreter is halted upon encountering an input() function until the user enters a value. Once a value is entered, the interpreter then proceeds to execute the next line.

Do it Yourself

Write a script that asks the user to enter the name of a protein in the terminal.

```
1 input('Please enter the name of a protein: ')
```

Variables And Types

We use variables to store data in the memory. Each variable has 3 characteristics: *scope*, *name*, and *type*. *Scope* and *name* must be mutually unique. Starting with *name*, we will discuss each of these characteristics in more details throughout this chapter.

Variable names

PEP–8 Naming Conventions

Name of a variable is in fact an alias for a location in the memory. You can think of it as a postbox, which is used as a substitute for an address. Similarly, we use variable names so we wouldn't have to use the actual address to the location we want in the memory because it would look something like `0x106fb8348`.

There are some relatively simple rules to follow when defining variable names, which ultimately boil down to:

Validity	Example	Note
Invalid	`5mins = 5 * SEC_IN_MIN`	Variables cannot start with numbers — raises a `SyntaxError`.
Invalid	`if = 2`	Attempts to overwrite an internal syntax — raises a `SyntaxError`.
Worst	`j = 2`	Overwrites the internal identifier for complex numbers.
Worst	`int = 2`	Overwrites the internal definition for integer numbers.
Bad	`a = 4`	Meaningless name + hard to identify in code.
Okay	`var1 = 0`	Meaningless name + hard to distinguish.
Better	`var_1 = 0`	Meaningless name.
Good	`current_state = 0`	Good name + easy to distinguish.
Okay	`five_mins_in_sec = 5 * 60`	Good name, vague value.
Good	`SEC_IN_MIN = 60`	Good name + good value + distinguished as a constant (conventionally, names made exclusively of capital letters signify constants).
Good	`five_mins2sec = 5 * SEC_IN_MIN`	Good name + good value.
Good	`five_mins2sec = 5 * 60  # 5 min * 60 sec`	Good name + vague value clarified by a comment.
Good	`current_protein = 'DNA Polymerase III'`	Good name + good value.

Remember

We should never overwrite an existing, built-in definition or identifier (e.g. `int` or `print`). We will be learning many such definitions and identifiers as we progress through this course. Nonetheless, the Jupyterlab notebook as well as any good IDE highlights syntaxes and built-in identifiers in different colours. In JupyterLab the default for built-in definitions is green. The exact colouring scheme depends on the IDE and the theme.

Once a variable is defined, its value may be altered or reset:

```
total_items = 2
print(total_items)
```

```
2
```

Variables containing integer numbers are known as **`int`**, and those containing decimal numbers are known as **`float`** in Python.

```
total_items = 3
print(total_items)
```

```
3
```

```
total_values = 3.2
print(total_values)
```

```
3.2
```

```
temperature = 16.
print(temperature)
```

```
16.0
```

Variables can contain characters as well; but to prevent Python from confusing them with meaningful commands, we use quotation marks. So long as we remain consistent, it doesn't matter whether we use single or double quotations. These variables are known as `string` or `str`:

```
forename = 'John'
surname = "Doe"


print('Hi,', forename, surname)
```

```
Hi, John Doe
```

Do it Yourself

Oxidised low-density lipoprotein (LDL) receptor 1 mediates the recognition, internalisation and degradation of oxidatively modified low density lipoprotein by vascular endothelial cells. Using the Universal Protein Resource (UniProt) website, find this protein for humans, and identify:

- UniProt entry number.
- Length of the protein (right at the top).
- Gene name (right at the top).

Store the information you retrieved, including the protein name, in 4 separate variables.

Display the values of these 4 variables in *one* line, and separate the items with 3 spaces, as follows:

```
Name EntryNo GeneName Length
```

```
name = 'Oxidised low-density lipoprotein (LDL) receptor 1'

uniprot_entry = 'P78380'

gene_name = 'OLR1'

length = 273

print(name, uniprot_entry, gene_name, length, sep='   ')
```

```
Oxidised low-density lipoprotein (LDL) receptor 1   P78380   OLR1   273
```

Do it Yourself

1. Write a script that upon execution, asks the user to enter the name of an enzyme and then retains the response in an appropriately named variable.
2. Use the variable to display an output similar to the following:

`ENZYME_NAME is an enzyme.`

where `ENZYME_NAME` is the name of the enzyme entered in the prompt.

3. Now alter your script to ask the user to enter the number of amino acids in that enzyme. Retain the value in another appropriately named variable.
4. Alter the output of your script to display a report in the following format:

`ENZYME_NAME is an enzyme containing a total number of AMINO_ACIDS} amino acids.`

where `AMINO_ACIDS` is the number of amino acids.

```
enzyme = input('Please enter the name of an enzyme: ')

print(enzyme, 'is an enzyme.')

length = input('How many amino acids does the enzyme contain? ')

print(enzyme, 'is an enzyme containing a total number of', length, '
    amino acids.')
```

Variable Types

Built-in Types

When it comes to types, programming languages may be divided into two distinct categories:

Types

- ☒ **Statically typed** language that require the programmer to define the type for every variable (statically typed).
- ☒ **Dynamically typed** languages that define and maintain the types on the fly.

Python is a dynamically typed language. This means that, unlike statically typed languages, we rarely need to worry about the *type* definitions because in the majority of cases, Python takes care of them for us.

Remember

In a dynamically typed language, it is the value of a variable that determines the type. This is because the types are determined on the fly by the Python interpreter as and when it encounters different variables and values.

Advanced Topic

In computer programming, type systems are syntactic methods to enforce and / or identify levels of abstraction. An entire field in computer science has been dedicated to the study of programming languages from a type–theoretic approach. This is primarily due to the implication of types and their underlying principles in such areas in software engineering as optimisation and security. To learn more about the study of type systems, refer to: Pierce B. Types and programming languages. Cambridge, Mass.: MIT Press; 2002.

Note

The values determine the type of a variable in dynamically typed languages. This is in contrast with statically typed languages where a variable must be initialised using a specific type before a value — whose type is consistent with the initialised variable, can be assigned to it.

Why learn about *types* in a dynamically typed programming language? Now that we know how to define variables in Python, you may have noticed that depending on the value, there are different forms of variables such as `int`, `float`, and `str`. These are built-in definitions known as variable *types*. In essence, types tell the computer how much space in the memory should be reserved for the value of a specific variable, and clarify the operations that may be applied to it.

Python enjoys a powerful type system out of the box. Table Built-in types in Python provides a comprehensive reference for the built-in types in Python. Built-in types are the types that exist in the language and do not require any third party libraries to implement or use.

Type	Type Name	Specification	Numeric	Sequence	Iterable	Mutable	Container
`bool`	Boolean	`True` or `False`	✔				
`int`	Integer	An integer number: $x \in \mathbb{Z}$	✔				
`float`	Floating point	A rational number: $x \in \mathbb{Q}$	✔				
`complex`	Complex	A complex number: $x \in \mathbb{C}$	✔				
`bytes`*	Bytes	Single byte					
`bytearray`*	Byte Array	Array of bytes; *i.e.* binary		✔	✔	✔	
`str`	String	Array of characters; *i.e.* text.		✔	✔		
`list`	List	Array of any combination		✔	✔	✔	
`tuple`	Tuple	Array of any combination		✔	✔		
`set`	Set	Collection of unique members		✔		✔	✔
`frozenset`*	Frozen Set	Collection of unique members		✔			✔
`dict`	Dictionary	Mapping (associative array)		✔	✔$	✔	✔

Figure 4: A comprehensive (but non-exhaustive) reference of built-in (native) types in Python 3. * Not discussed in this course — included for reference only. $ `dict` is not an iterable by default, however, it is possible to iterate through its keys. **Mutability** is an important concept in programming. A mutable object is an object whose value(s) may be altered. This will become clearer once we study `list` and `tuple`. Find out more about mutability in Python from the documentations}. **Complex numbers** refer to a set of numbers that have a real part, and an imaginary part; where the imaginary part is defined as $\sqrt{-1}$. These numbers are very useful in the study of oscillatory behaviours and flow (e.g. heat, fluid, electricity). To learn more about complex numbers, watch this Khan Academy video tutorial.

Sometimes we might need want to know what is the type of a variable. To do so, we use the build-in function type() as follows:

```
1 total_items = 2
2
3 print(type(total_items))
```

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

```
1 total_values = 3.2
```

```
2
3 print(type(total_values))
```

```
1 <class 'float'>
```

```
1 temperature = 16.
2
3 print(type(temperature))
```

```
1 <class 'float'>
```

```
1 phase = 12.5+1.5j
2
3 print(type(phase))
```

```
1 <class 'complex'>
```

```
1 full_name = 'John Doe'
2
3 print(type(full_name))
```

```
1 <class 'str'>
```

Remember

In Python, a variable / value of a certain type may be referred to as an *instance* of that type. For instance, an integer value whose type in Python is defined as *int* is said to be an **instance of type `int`**.

Do it Yourself

Determine and display the type for each of these values:

- 32
- 24.3454
- 2.5 + 1.5
- “RNA Polymerase III”
- 0
- .5 - 1
- 1.3e-5
- 3e5

The result for each value should be represented in the following format:

```
Value X is an instance of <class 'Y'>
```

```
value = 32

value_type = type(value)

print('Value', value, 'is an instance of', value_type)
```

```
Value 32 is an instance of <class 'int'>
```

```
value = 24.3454

value_type = type(value)

print('Value', value, 'is an instance of', value_type)
```

```
Value 24.3454 is an instance of <class 'float'>
```

```
value = 2.5 + 1.5

value_type = type(value)

print('Value', value, 'is an instance of', value_type)
```

```
Value 4.0 is an instance of <class 'float'>
```

```
value = "RNA Polymerase III"

value_type = type(value)

print('Value', value, 'is an instance of', value_type)
```

```
Value RNA Polymerase III is an instance of <class 'str'>
```

```
value = 0

value_type = type(value)

print('Value', value, 'is an instance of', value_type)
```

```
Value 0 is an instance of <class 'int'>
```

```
value = .5 - 1

value_type = type(value)

print('Value', value, 'is an instance of', value_type)
```

```
Value -0.5 is an instance of <class 'float'>
```

```
value = 1.3e-5

value_type = type(value)

print('Value', value, 'is an instance of', value_type)
```

```
Value 1.3e-05 is an instance of <class 'float'>
```

```
value = 3e5

value_type = type(value)

print('Value', value, 'is an instance of', value_type)
```

```
Value 300000.0 is an instance of <class 'float'>
```

Conversion of types

Why convert types?

It is sometimes necessary to have the values returned by the input() function — *i.e.* the user's response, in other types. Imagine the following scenario:

"We ask our user to enter the total volume of their purified protein, so that we can work out the amount of assay they need to conduct a specific experiment. To calculate this assay volume using the volume of the purified protein, we need to perform mathematical calculations based on the response we receive from our user. It is not possible to perform mathematical operations on non-numeric values. Therefore, we ought to somehow convert the type from `str` to a numeric type."

The possibility of converting from one type to another depends entirely on the *value*, the *source type*, and the *target type*. For instance; we can convert an instance of type `str` (source type) to one of type **`int`** (target type) if and only if the source value consists entirely of numbers and there are *no* other characters.

Remember

To convert a variable from one type to another, we use the *Type Name* of the target type (as described in Table Built-in types in Python and treat it as a function.

For instance, to convert a variable to integer, we:

- look up the *Type Name* for integer from Table Built-in types in Python
- then treat the *Type Name* as a function: `int()`
- use the function to convert our variable: new_var = `int`(old_var)

Here is an example of how we convert types in Python:

```
value_a = '12'

print(value_a, type(value_a))
```

```
12 <class 'str'>
```

```
value_b = int(value_a)

print(value_b, type(value_b))
```

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

If we attempt to convert a variable that contains non-numeric values, a `ValueError` is raised:

```
value_a = '12y'

print(value_a, type(value_a))
```

```
12y <class 'str'>
```

```
value_b = int(value_a)
```

```
Error in py_call_impl(callable, dots$args, dots$keywords): ValueError:
    invalid literal for int() with base 10: '12y'

Detailed traceback:
  File "<string>", line 1, in <module>
```

Do it Yourself

In programming, we routinely face errors resulting from different mistakes. The process of finding and correcting such mistakes in the code is referred to as *debugging*.

We have been given the following snippet written in Python 3:

```
value_a = 3
value_b = '2'

result = value_a + value_b
print(value_a, '+', value_b, '=', result)
```

but when the code is executed, we encounter an error message as follows:

```
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Debug the snippet so that the correct result is displayed:

```
3 + 2 = 5
```

```
value_a = 3

value_b = '2'

result = value_a + int(value_b)

print(value_a, '+', value_b, '=', result)
```

```
3 + 2 = 5
```

Handling input variables

:::::::::::::::::::::::::::::::: discussion

When we use input() to obtain a value from the user, the results are by default an instance of type `str`. An input() function always stores the response as a `str` value, no matter what the user enters. However, it is possible to convert the type afterwards.

::::::::::::::::::::::::::::::::

Remember

The input() function *always* returns a value of type `str` regardless of the user's response. In other words, if a user's response to an input() request is numeric, Python will *not* automatically recognise it as a numeric type.

We may use *type conversion* in conjunction with the values returned by the input() function:

```
response = input('Please enter a numeric value: ')

response_numeric = float(response)

print('response:', response)
print('response type:', type(response))
print('response_numeric:', response_numeric)
print('response_numeric type:', type(response_numeric))
```

The output shows the results when we enter numeric values as directed.

Do it Yourself

We know that each amino acid in a protein is encoded by a triplet of mRNA nucleotides.

With that in mind, alter the script you wrote for Do it Yourself and use the number of amino acids entered by the user to calculate the number of mRNA nucleotides.

Display the results in the following format:

```
ENZYME_NAME is an enzyme with AMINO_ACIDS amino acids and NUCLEOTIDES
 nucleotides.
```

where `NUCLEOTIDES` is the total number of mRNA nucleotides that you calculated.

Note: Multiplication is represented using the asterisk (`*`) sign.

```
1 enzyme = input('Please enter the name of an enzyme: ')
2
3 length = input('How many amino acids does the enzyme contain? ')
4
5 nucleotides = 3 * int(length)
6
7 print(enzyme, 'is an enzyme with', length, 'amino acids and',
     nucleotides, 'nucleotides.')
```

Variable scopes

Resolution of names

When defining a variable, we should always consider where in our programme we intent to use it. The more localised our variables, the better. This is because local variables are easier to distinguish, and thus reduce the chance of making mistakes — e.g. unintentionally redefine or alter the value of an existing variable.

To that end, the scope of a variable defines the ability to reference a variable from different *points* in our programmes. The concept of local variables becomes clearer once we explore functions in programming in chapter Functions.

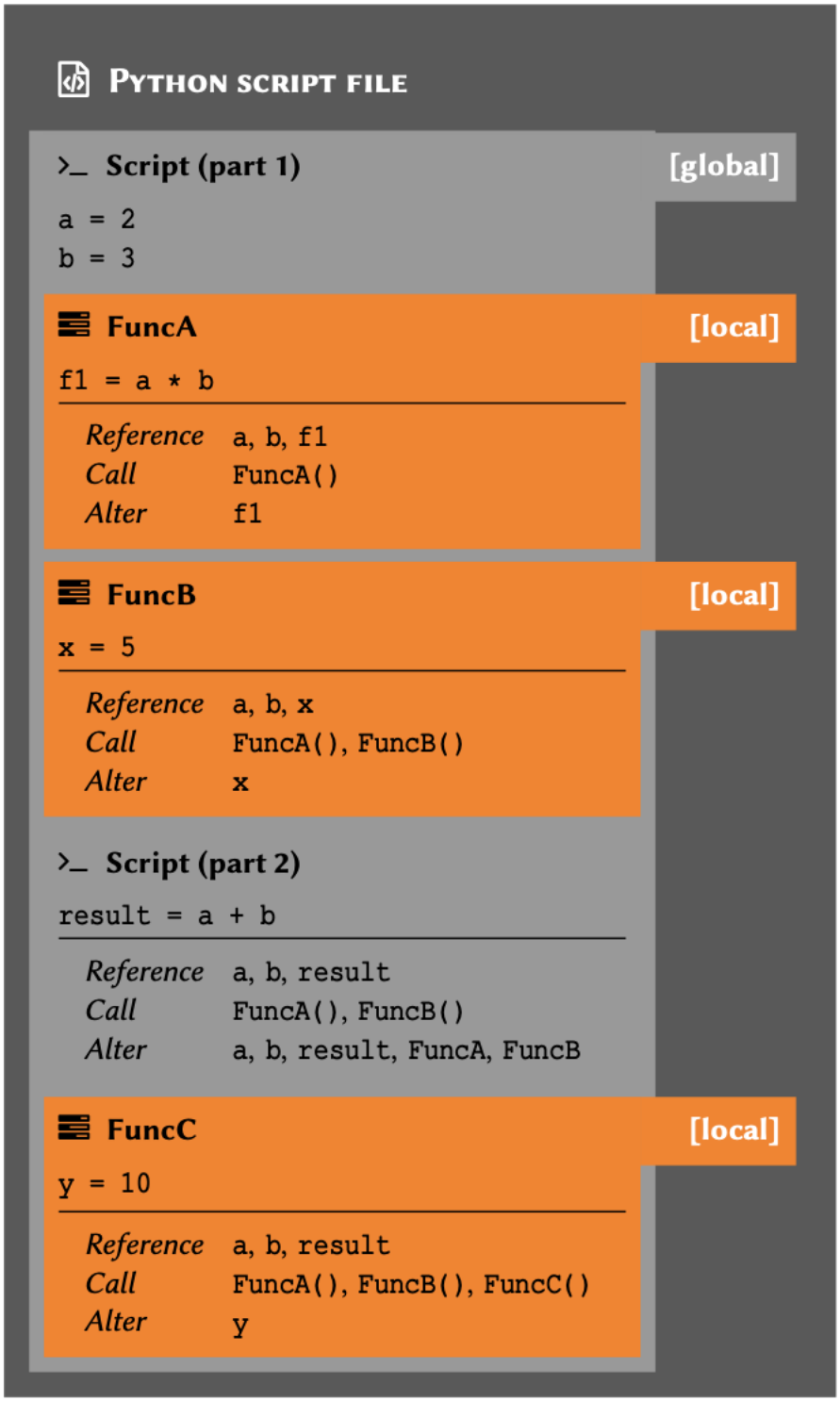
As displayed in Figure Variable scopes, the point *at* or *from* which a variable can be referenced depends on the location where the variable is defined.

In essence, there are three general rules to remember in relation variable scopes in Python:

I. A variable that is defined in the outer scope, can be *accessed* or *called* in the inner scopes, but it cannot be *altered* implicitly. Not that such variables may still be altered using special techniques (not discussed).

II. A variable that is defined in the innermost scopes (local), can only be *accessed*, *called*, or *altered* within the boundaries of the scope it is defined in.

III. The inner scopes *from* which a variable is referenced must themselves have be contained within the defining scope — e.g. in `FuncB` of Figure Variable scopes, we can reference `a`, `b`, and `x`; but not `f1`. This is because the scope of `f1` is `Script` → `FuncA`, so it can only be referenced from `Script` → `FuncA` → …, but not ‘`Script`→ … or `Script` → `FuncB` → ….



Python is an interpreted language. This means that the Python interpreter goes through the codes that we write line by line, interpreting it to machine language. It is only then that the commands are

processed and executed by the computer. On that account, a variable (or a function) can be referenced only *after* its initial definition. That is why, for instance, in `Script (part 2)` of Figure Variable scopes, we can reference every variable and function except for `FuncC`, which is declared further down the code hierarchy.

Although scope and hierarchy appear at first glance as theoretical concepts in programming, their implications are entirely practical. The definition of these principles vary from one programming language to another. As such, it is essential to understand these principles and their implications in relation to any programming language we are trying to learn.

Optional: How to use terminal environment?

How to navigate the terminal in Mac OS X and Linux?

The Unix terminal — *i.e.* the one used by Mac OS X® and Linux; uses a command language known as Bash, also referred to as the Unix Shell. This is not the topic of this book, so we won't discuss its properties and features in much detail.

Microsoft Windows®, on the other hand, relies on DOS (Microsoft Disk Operating System®) commands to navigate the terminal. The terminal environment in is sometimes referred to as the `CMD` or the command prompt.

The enviroment in which we can use Bash or DOS commands is known as the terminal. To use a terminal, we need a terminal emulator. Every operating system has a default terminal emulator, but you can always use an alternative software if you so wish.

Current working directory

To navigate the terminal environment, we should first find out where we are. Here is how we do that:

Unix	Microsoft Windows®
`pwd` ⏎	`pwd` ⏎

Remember that the path to an enviroment is displayed (and supplied differently in Unix and Microsoft Windows®). To that end, the response (output) to these commands may slightly vary:

Unix	Microsoft Windows®
`/home/UserName/Documents`	`C:\Documents`

Notice that in Microsoft Windows®, we separate directories using a backslash (\ ), whilst in Unix, we use a foreslash (/) for that purpose.

Changing the current directory

To navigate to another directory, we do as follows:

	Unix	Microsoft Windows®
Without spaces in the name	`cd /home/UserName/Documents/Python_Scripts` ⏎ ⏎	`cd C:\Documents\Python_Scripts` ⏎
With spaces in the name	`cd "/home/UserName/Documents/Python Scripts"` ⏎	`cd "C:\Documents\Python Scripts"` ⏎

Note that paths containing one or more space characters must be encapsulated by double quotation marks (`"..."`).

If we are already in the `Documents` directory, and want to navigate onto `Python_Scripts`, we can use the following shorthand versions of the above method:

	Unix	Microsoft Windows®
Without spaces in the name	`cd Python_Scripts` ⏎	`cd Python_Scripts` ⏎
With spaces in the name	`cd "Python Scripts"` ⏎	`cd "Python Scripts"` ⏎

Parent directory

To navigate to the parent directory — *i.e.* from `Python_Scripts` to `Documents` in this example; we do:

Unix	Microsoft Windows®

Home directory

We can also navigate all the way back to the home directory in one go:

Unix	Microsoft Windows®
`cd` ⏎	`cd \` ⏎

Creating a new directory

We can create a new directory as follows:

Unix	Microsoft Windows®
`mkdir new_dir_name` ⏎	`md new_dir_name` ⏎

Contents of a directory

To see the contents of a directory, we use the following commands:

Unix	Microsoft Windows®
`ls` ⏎	`dir` ⏎

Operations

Through our experimentations with variable types, we already know that variables may be subject to different operations.

When assessing type conversions we also established that the operations we can apply to each variable depend on the *type* of that variable. To that end, we learned that although it is sometimes possible to mix variables from different types to perform an operation — e.g. multiplying a floating point number with an integer, there are some logical restrictions in place.

Throughout this section, we will take a closer look into different types of operations in Python. This will allow us to gain a deeper insight into the concept and familiarise ourselves with the underlying logic.

To recapitulate on what we have done so far, we start off by reviewing *additions* — the most basic of all operations.

Give the variable `total_items`:

```
1 total_items = 2
2
```

```
print(total_items)
```

```
2
```

We can increment the value of an *existing* variable by 1 as follows:

```
total_items = total_items + 1

print(total_items)
```

```
3
```

Given 2 different variables, each containing a different value; we can perform an operation on these values and store the result in *another* variable without altering the original variables in any way:

```
old_items = 4
new_items = 3

total_items = old_items + new_items

print(total_items)
```

```
7
```

We can change the value of an *existing* variable using the value stored in *another* variable:

```
new_items = 5
total_items = total_items + new_items

print(total_items)
```

```
12
```

There is also a shorthand method for applying the operation on an *existing* variable:

```
total_items = 2

print(total_items)
```

```
2
```

```
total_items += 1

print(total_items)
```

```
3
```

```
new_items = 5
```

```
total_items += new_items

print(total_items)
```

```
8
```

As highlighted in the introduction, different operations may be applied to any variable or value. Throughout the rest of this section, we will explore the most fundamental operations in programming, and learn about their implementation in Python.

Remember

There are 2 very general categories of operations in programming: *mathematical*, and *logical*. Naturally, we use mathematical operations to perform calculations, and logical operations to perform tests.

Mathematical Operations

Suppose a and b are 2 variables representing integer numbers as follows:

```
a = 17
b = 5
```

Using a and b we can itemise built-in mathematical operations in Python as follows:

Operation	Notation	Result
Addition	`a + b`	22
Subtraction	`a - b`	12
Multiplication	`a * b`	85
Division	`a / b`	3.4
Floor quotient	`a // b`	3
Remainder	`a % b`	2
Quotient and remainder	`divmod(a, b)`	(3, 2)
Power	`a ** b`	1419857
Absolute value	`abs(b - a)`	12

Figure 5: Routine mathematical operations in Python

Remember

As far as mathematical operations are concerned, variables a and b may be an instance of any *numeric* type. See Table Routine mathematical operations in Python to find out more about numeric types in Python.

Values of type `int` have been chosen in our examples to facilitate the understanding of the results.

Do it Yourself

1. Calculate the following and store the results in appropriately named variables:

a. 5.8×3.3
b. $\frac{180}{6}$
c. $35 - 3.0$
d. $35 - 3$
e. 2^{1000}

Display the result of each calculation – including the type, in the following format:

```
Result: X is an instance of <class 'Y'>
```

2. Now using the results you obtained:
   I. Can you explain why is the result of $35 - 3.0$ is an instance of type `float`, whilst that of $35 - 3$ is of type `int`?

II. Unlike the numeric types, string values have a length. To obtain the length of a string value, we use len(). Convert the result for 2^{1000} from `int` to `str`, then use the aforementioned function to work out the length of the number — i.e. how many digits is it made of?

If you feel adventurous, you can try this for 2^{10000} or higher; but beware that you might overwhelm your computer and need a restart it if you go too far (i.e. above $2^{1000000}$). Just make sure you save everything beforehand, so you don't accidentally step on your own foot.}

Hint: We discuss len() in subsection of arrays. However, at this point, you should be able to use the official documentations and StackOverflow to work out how it works.

```
1 q1_a = 5.8 * 3.3
2 print('Result:', q1_a, 'is an instance of', type(q1_a))
```

```
1 Result: 19.139999999999997 is an instance of <class 'float'>
```

```
1 q1_b = 180 / 6
2 print('Result:', q1_b, 'is an instance of', type(q1_b))
```

```
Result: 30.0 is an instance of <class 'float'>
```

```
q1_c = 35 - 3.0
print('Result:', q1_c, 'is an instance of', type(q1_c))
```

```
Result: 32.0 is an instance of <class 'float'>
```

```
q1_d = 35 - 3
print('Result:', q1_d, 'is an instance of', type(q1_d))
```

```
Result: 32 is an instance of <class 'int'>
```

```
q1_e = 2 ** 1000
print('Result:', q1_e, 'is an instance of', type(q1_e))
```

```
Result:
    107150860718626732094842504906000181056140481170553360744375038837035105112493612
     is an instance of <class 'int'>
```

In the case of $35 - 3.0$ vs $35 - 3$, the former includes a floating point number. Operations involving multiple numeric types always produce the results as an instance of the type that covers all of the operands – i.e. **`float`** covers **`int`**, but not vice-versa.

```
big_num = 2 ** 1000
big_num_str = str(big_num)
big_num_len = len(big_num_str)

print('Length of 2**1000:', big_num_len)
```

```
Length of 2**1000: 302
```

Interesting Fact

As of Python 3.6, you can use an underscores (_) *within* large numbers as a separator to make them easier to read in your code. For instance, instead of `x = 1000000`, you can write `x = 1_000_000`.

Shorthands When it comes to mathematical operations in Python, there is a frequently used shorthand method that every Python programmer should be familiar with.

Suppose we have a variable defined as `total_residues = 52` and want to perform a mathematical operation on it. However, we would like to store the result of that operation in `total_residues` instead of a new variable. In such cases, we can do as follows:

```
total_residues = 52

# Addition:
total_residues += 8

print(total_residues)
```

```
60
```

```
# Subtraction:
total_residues -= 10

print(total_residues)
```

```
50
```

```
# Multiplication:
total_residues *= 2

print(total_residues)
```

```
100
```

```
# Division:
total_residues /= 4

print(total_residues)
```

```
25.0
```

```
# Floor quotient:
total_residues //= 2

print(total_residues)
```

```
12.0
```

```
# Remainder:
total_residues %= 5

print(total_residues)
```

```
2.0
```

```
# Power:
total_residues **= 3

```

```
print(total_residues)
```

```
8.0
```

We can also perform such operations using multiple variables:

```
total_residues = 52
new_residues = 8
number_of_proteins = 3

total_residues += new_residues

print(total_residues)
```

```
60
```

```
total_residues += (number_of_proteins * new_residues)

print(total_residues)
```

```
84
```

Do it Yourself

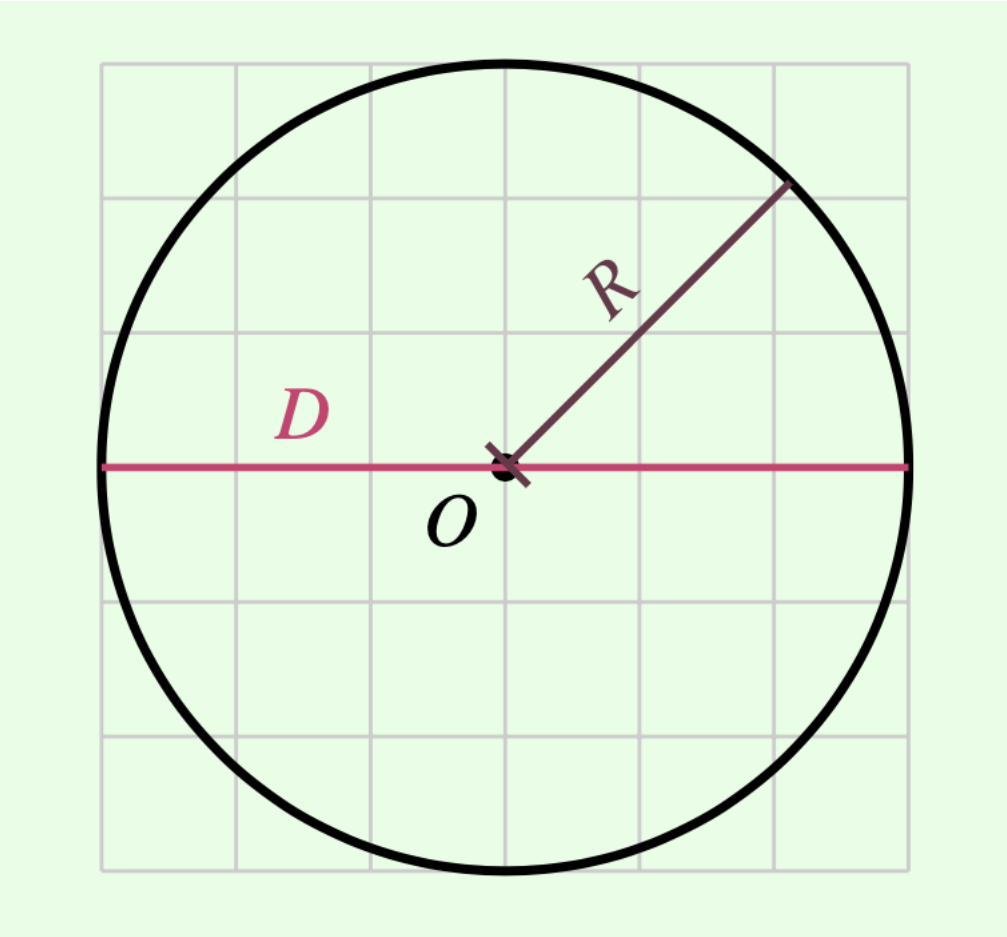
1. Given:

    - Circumference: $C = 18.84956$
    - Radius: $R = 3$

and considering that the properties of a circle are defined as follows:

$$\pi = \frac{C}{D}$$

calculate π using the above equation and store it in a variable named `pi`:



Then round the results to 5 decimal places and display the result in the following format:

```
The value of pi calculated to 5 decimal places: X.XXXXX
```

Note: To round floating point numbers in Python, we use round(). This is a built-in function that takes 2 input arguments: the first is the variable/value to be rounded, and the second is the number decimal places. Read more about round() in the official documentations.

2. Now without creating a new variable, perform the following operation:

$$pi = \frac{pi}{(3 \bmod 2) - 1}$$

where the expression "$3 \bmod 2$" represents the remainder for the division of 3 by 2.

Explain the output.

```
c = 18.84956
r = 3
```

```
d = r * 2

pi = c / d

print('The value of pi calculated to 5 decimal places:', round(pi, 5))
```

```
The value of pi calculated to 5 decimal places: 3.14159
```

```
    pi /= (3 % 2) - 1
```

The calculation raises a `ZeroDivisionError`. This is because division by zero is mathematically impossible.

Precedence In mathematics and computer programming, there is a collection of conventional rules on the precedence of procedures to evaluate a mathematical expression. This collection of rules is referred to as the *order of operation* or *operator precedence*.

Suppose we have a mathematical expression as follows:

$$x = 2 + 3 \times 9$$

Such an expression can *only* be evaluated correctly if we do the multiplication first and then perform the addition. This means that the evaluation is done as follows:

$$given : 3 \times 9 = 27$$

$$\implies x = 2 + 27$$

$$= 29$$

For instance, in an expression such as:

$$x = 2 \times (3 + (5 - 1)^2)$$

the evaluation workflow may be described as follows:

$$x = 2 \times (3 + 4^2)$$

$$= 2 \times (3 + 16)$$

$$= 38$$

The same principle applies in Python. This means that if we use Python to evaluate the above expression, the result would be identical:

```
result = 2 * (3 + (5 - 1) ** 2)

print(result)
```

```
38
```

Remember

Operator precedence in mathematical operations may be described as follows:

1. Exponents and roots
2. Multiplication and division
3. Addition and subtraction

If there are any parenthesis ( ) in the expression, the expression is evaluated from the innermost parenthesis outwards.

Do it Yourself

Display the result of each item in the following format:

```
EXPRESSION = RESULT
```

For example:

```
    2 + 3 = 5
```

1. Calculate each expression *without* using parentheses:

a. $3 \times \frac{2}{4}$
b. $5 + 3 \times \frac{2}{4}$
c. $3 \times \frac{2}{4} + 5$
d. $\frac{2}{4} \times 3$

2. Calculate these expressions *using* parentheses:

a. $5 + \frac{2}{4} \times 3$
b. $5 + \frac{2 \times 3}{4}$

c. $5 + \frac{2}{4 \times 3}$

3. Given

```
a = 2
b = 5
```

use a and b to calculate the following expressions:

a. $(a + b)^2$
b. $a^2 + 2ab + b^2$

```
q1_a = 3 * 2 / 4
print('3 * 2 / 4 =', q1_a)
```

```
3 * 2 / 4 = 1.5
```

```
q1_b = 5 + 3 * 2 / 4
print('5 + 3 * 2 / 4 =', q1_b)
```

```
5 + 3 * 2 / 4 = 6.5
```

```
q1_c = 3 * 2 / 4 + 5
print('3 * 2 / 4 + 5 =', q1_c)
```

```
3 * 2 / 4 + 5 = 6.5
```

```
q1_d = 2 / 4 * 3
print('2 / 4 * 3 =', q1_d)
```

```
2 / 4 * 3 = 1.5
```

```
q2_a = 5 + (2 / 4) * 3
print('5 + (2 / 4) * 3 =', q2_a)
```

```
5 + (2 / 4) * 3 = 6.5
```

```
q2_b = 5 + (2 * 3) / 4
print('5 + (2 * 3) / 4 =', q2_b)
```

```
5 + (2 * 3) / 4 = 6.5
```

```
q2_c = 5 + 2 / (4 * 3)
print('5 + 2 / (4 * 3) =', q2_c)
```

```
5 + 2 / (4 * 3) = 5.166666666666667
```

```
a = 2
b = 5

q3_a = (a + b) ** 2
print('(a + b)^2 =', q3_a)
```

```
(a + b)^2 = 49
```

```
q3_b = a ** 2 + 2 * a * b + b ** 2
print('a^2 + 2ab + b^2 =', q3_b)
```

```
a^2 + 2ab + b^2 = 49
```

Non-numeric values It sometimes makes sense to apply *some* mathematical operations to non-numeric variables too.

We can multiply strings to repeat them. There is no specific advantage to the use of multiplication instead of manually repeating characters or words, but it makes our code look cleaner, and that's always a good thing!

We can also add string values to each other. This is called *string concatenation*. It is a useful method for concatenating a few strings and / or string variables.

```
SEPARATOR = '-' * 20
NEW_LINE = '\n'
SPACE = ' '

forename = 'Jane'
surname = 'Doe'
birthday = '01/01/1990'

full_name = forename + SPACE + surname

data = full_name + NEW_LINE + SEPARATOR + NEW_LINE + 'DoB: ' + birthday

print(data)
```

```
Jane Doe
--------------------
DoB: 01/01/1990
```

Remember

New line character or `'\n'` is a universal directive to induce a line-break in Unix based operating systems (MACOS) and Linux). In WINDOWS, we usually us `'\r'` or `'\r\n'` instead. These are known as

escape sequences, which we explore in additional details under string operations in chapter Strings

Do it Yourself

The risk of Huntington's disease appears to increase proportional to the continuous repetition of `CAG` nucleotides (glutamine codon) once they exceed 35 near the beginning of the Huntingtin (`IT15`) gene. The `CAG` repeats are also referred to as a polyglutamine or polyQ tract.

```
glutamine_codon = 'CAG'
```

1. Create a polynucleotide chain representing 36 glutamine codons. Store the result in a variable called `polyq_codons`.

Display the result as:

```
Polyglutamine codons with 36 repeats: XXXXXXXXX...
```

2. Use len() to work out the length of `polyq_codons`, and store the result in a variable called `polyq_codons_length`.

Display the result in the following format:

```
Number of nucleotides in a polyglutamine with 36 repeats: XXX
```

3. Use len() to work out the length of `glutamin_codon`, and store the result in variable `amino_acids_per_codon`.
4. Divide `polyq_codons_length` by `amino_acids_per_codon` to prove that the chain contains the codon for exactly 36 amino acids. Store the result in variable `polyq_peptide_length`.

Display the result in the following format:

```
Number of amino acids in a polyglutamine with 36 repeats: XXX
```

5. Determine the types for the following variable:

- amino_acids_per_codon
- polyq_codons_length
- polyq_peptide_length

and display the result for each item in the following format:

```
Value: XXX - Type: <class 'XXXX'>
```

6. Are all the variables in task #5 of the same type? Why?
7. Repeat from task #4, but this time use an alternative method of division as outlined in See Table Routine mathematical operations in Python.

```
glutamine_codon = 'CAG'

polyq_codons = glutamine_codon * 36

print('Polyglutamine codons with 36 repeats:', polyq_codons)
```

```
Polyglutamine codons with 36 repeats:
    CAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCA
```

```
polyq_codons_length = len(polyq_codons)

print('Number of nucleotides in a polyglutamine with 36 repeats:',
    polyq_codons_length)
```

```
Number of nucleotides in a polyglutamine with 36 repeats: 108
```

```
amino_acids_per_codon = len(glutamine_codon)
```

```
polyq_peptide_length = polyq_codons_length / amino_acids_per_codon

print('Number of amino acids in a polyglutamine with 36 repeats:',
    polyq_peptide_length)
```

```
Number of amino acids in a polyglutamine with 36 repeats: 36.0
```

```
print('Value:', amino_acids_per_codon, '- Type:', type(
    amino_acids_per_codon))

print('Value:', polyq_codons_length, '- Type:', type(
    polyq_codons_length))

print('Value:', polyq_peptide_length, '- Type:', type(
    polyq_peptide_length))
```

```
Value: 3 - Type: <class 'int'>
Value: 108 - Type: <class 'int'>
Value: 36.0 - Type: <class 'float'>
```

No, `polyq_peptide_length` is an instance of type **`float`**. This is because we used the normal division (`/`) and not floor division (`//`}) to calculate its value. The result of normal division is always presented as a floating point number.

```
polyq_peptide_length = polyq_codons_length // amino_acids_per_codon

print('Number of amino acids in a polyglutamine with 36 repeats:',
    polyq_peptide_length)

print('Value:', amino_acids_per_codon, '- Type:', type(
    amino_acids_per_codon))

print('Value:', polyq_codons_length, '- Type:', type(
    polyq_codons_length))

print('Value:', polyq_peptide_length, '- Type:', type(
    polyq_peptide_length))
```

```
Number of amino acids in a polyglutamine with 36 repeats: 36
Value: 3 - Type: <class 'int'>
Value: 108 - Type: <class 'int'>
Value: 36 - Type: <class 'int'>
```

Interesting Fact

The Boolean data type is named after the English mathematician and logician George Boole (1815–1864).

Logical Operations

An operation may involve a comparison. The result of such operations is either `True` or `False`. This is known as the *Boolean* or `bool` data type. In reality, however, computers record `True` and `False` as `1` and `0` respectively.

Operations with Boolean results are referred to as *logical operations*. Testing the results of such operations is known as *truth value testing*.

Given the two variables `a` and `b` as follows:

```
a = 17
b = 5
```

Boolean operations may be defined as outlined in Table Routine logical operations in Python..

Do it Yourself

We know that in algebra, the first identity (square of a binomial) is:

Logic	Statement	Result
Equivalence	`a == b`	`False`
Non-equivalence	`a != b`	`True`
Greater	`a > b`	`True`
Either greater or equal	`a >= b`	`True`
Smaller	`a < b`	`False`
Either smaller or equal	`a <= b`	`False`
Between	`5 < a < 20`	`True`

Table 2.2 Routine logical operations in Python.

Figure 6: Routine logical operations in Python.

$$(a+b)^2 = a^2 + 2ab + b^2$$

now given:

```
a = 15
b = 4
```

1. Calculate

$$y_1 = (a+b)^2$$

$$y_2 = a^2 + 2ab + b^2$$

Display the results in the following format:

```
y1 = XX
y2 = XX
```

2. Determine whether or not `y_1` is indeed equal to `y_2`. Store the result of your test in another variable called `equivalence`. Display the results in the following format:

```
Where a = XX and b = XX:
y1 is equal to y2: [True/False]
```

```
a = 15
b = 4

y_1 = (a + b) ** 2
y_2 = a ** 2 + 2 * a * b + b ** 2
```

```
6
7 print('y1 =', y_1)
8 print('y2 =', y_2)
```

```
1 y1 = 361
2 y2 = 361
```

```
1 equivalence = y_1 == y_2
2
3 print('Where a =', a, ' and b=', b)
4 print('y1 is equal to y2:', equivalence)
```

```
1 Where a = 15  and b= 4
2 y1 is equal to y2: True
```

Negation We can also use negation in logical operations. Negation in Python is implemented using not:

Logic	Statement	Result
Not equal	`not a == b`	`True`
Not greater	`not a > b`	`False`
Neither greater nor equal	`not a >= b`	`False`
Smaller	`not a < b`	`True`
Neither smaller nor equal	`not a <= b`	`True`

Figure 7: Negations in Python.

Do it Yourself

Using the information from previous Do it Yourself:

1. Without using not, determine whether or not `y_1` is *not equal* to `y_2`. Display the result of your test and store it in another variable called `inequivalent`.
2. Negate `inequivalent` and display the result.

```
1 inequivalent = y_1 != y_2
2
3 print(inequivalent)
```

```
1 False
```

```
inequivalent_negated = not inequivalent

print(inequivalent_negated)
```

```
True
```

Disjunctions and Conjunctions Logical operations may be combined using conjunction with and and disjunction with or to create more complex logics:

Logic	Statement	Result
Either smaller or equal	`a < b or a == b`	`False`
Either greater or smaller	`a < b or a > b`	`True`
Either greater or equal	`a > b or a == b`	`True`
Greater but not greater than	`a > b and not a > 18`	`True`
Either equal or not between	`a == b or not b < a < 20`	`False`
Equal and between	`a == b and 5 <= a < 20`	`False`

Figure 8: Disjunctions and Conjunctions in Python.

Do it Yourself

Given

```
a = True
b = False
c = True
```

Evaluate the following statements:

1. a == b
2. a == c
3. a `or` b
4. a `and` b
5. a `or` b `and` c
6. (a `or` b) `and` c
7. `not` a `or` (b `and` c)
8. `not` a `or` `not`(b `and` c)
9. `not` a `and` `not`(b `and` c)
10. `not` a `and` `not`(b `or` c)

Display the results in the following format:

```
1. [True/False]
2. [True/False]
        ...
```

Given that:

```
a = True
b = False
c = True
```

```
print('1.', a == b)
```

```
1. False
```

```
print('2.', a == c)
```

```
2. False
```

```
print('3.', a or b)
```

```
3. 15
```

```
print('4.', a and b)
```

```
4. 4
```

```
print('5.', a or b and c)
```

```
5. 15
```

```
print('6.', (a or b) and c)
```

```
6. 18.84956
```

```
print('7.', not a or (b and c))
```

```
7. 18.84956
```

```
print('8.', not a or not(b and c))
```

```
8. False
```

```
print('9.', not a and not(b and c))
```

```
9. False
```

```
print('10.', not a and not(b or c))
```

```
10. False
```

Complex logical operations It may help to break down more complex operations, or use parenthesis to make them easier to both read and write:

Logic	Statement	Result
Complex logic	`a == b or (5 <= a or b < 20 and a < b)`	`True`
Complex logic	`a == b or (5 <= a or b < 20) and a < b`	`False`
Complex logic	`a == b or 5 <= a or (b < 20 and a < b)`	`True`

Figure 9: Complex Logical Operations in Python.

Notice that in the last example, all notations are essentially the same and only vary in terms of their collective results as defined using parenthesis. Always remember that in a logical statement:

Logical statement

- ☒ The statement in parenthesis does **not** have precedence over the rest of the state (unlike mathematical statements). It merely defines an independent part of the operation whose response is evaluated separately.
- ☒ The precedence is established on a first come, first serve basis (from left to right).
- ☒ Always use parenthesis in longer statements for clarification.
- ☒ In disjunctive statements —i.e. a > 5 `or` b > 5, if the first part is `True`, the second part is *not* checked. In other words, if a is greater than 5, the computer does not proceed to check whether or not b is greater than 5.
- ☒ In conjunctive statements —i.e. a > 5 `and` b > 5, the statement proceeds to the seconds part if and only if the first part is `True`. In other words, the result of a conjunctive statement is only `True` if and only if both a and b are greater than 5. If a is `False`, the entire statement will inevitably be `False`.
- ☒ The longer the statement, the more difficult it would be to understand it properly, and by extension, the more likely it would be to cause problems.

```
a, b, c = 17, 5, 2  # Alternative method to define variables.
```

```
# Disjunction: false OR true.
a < b or b > c
```

```
True
```

```
# Disjunction: true OR true.
a > b or b > c
```

```
True
```

```
# Conjunction: true AND true.
a > b and b > c
```

```
True
```

```
# Conjunction: false and true.
a < b and b > c
```

```
False
```

```
# Disjunction and conjunction: true OR false AND true
a > b or b < c and b < a
```

```
True
```

```
# Disjunction and conjunction: false OR true AND false
a < b or b > c and b > a
```

```
False
```

```
# Disjunctions and conjunction: false OR true AND true
a < b or b > c and b < a
```

```
True
```

```
# Disjunction and negated conjunction and conjunction:
# true AND NOT false AND false
a < b or not b < c and b > a
```

```
False
```

```
# Disjunction and negated conjunction - similar to the
# previous example: true AND NOT (false AND false)
a < b or not (b < c and b > a)
```

```
True
```

These are only a few examples. There are endless possibilities, try them yourself and see how they work.

Remember

Some logical operations may be written in different ways. However, we should always use the notation that is most coherent in the context of our code. If in doubt, use the simplest / shortest notation.

To that end, you may want to use variables to split complex statements down to smaller portions:

```
age_a, age_b = 15, 35

are_positive = age_a > 0 and age_b > 0

a_is_older = are_positive and (age_a > age_b)
b_is_older = are_positive and (age_a < age_b)

a_is_teenager = are_positive and 12 < age_a < 20
b_is_teenager = are_positive and 12 < age_b < 20

a_is_teenager and b_is_older
```

```
True
```

```
a_is_teenager and a_is_older
```

```
False
```

```
a_is_teenager and (b_is_teenager or b_is_older)
```

```
True
```

Do it Yourself

Given

```
a = 3
b = 13
```

Test the following statements and display the results:

- $a^2 < b$
- $3 - a^3 < b$
- $|25 - a^2| > b$
- $25 \bmod a^2 > b$
- $25 \bmod a^2 > b$ or $25 \bmod b < a$
- $25 \bmod a^2 < b$ and $25 \bmod b > a$
- $\frac{12}{a}$ and $a \times 4 < b$

where "|...|" represents the absolute value, and "$n \bmod m$" represents the remainder for the division of n by m.}

Display the results in the following format:

```
1. [True/False]
2. [True/False]
...
```

```
#Given that:
a = 3
b = 13
print('1.', a**2 < b)
```

```
1. True
```

```
print('2.', (3 - a**3) < b)
```

```
2. True
```

```
print('3.', abs(25 - a**2) > b)
```

```
3. True
```

```
print('4.', (25 % a**2) > b)
```

```
4. False
```

```
print('5.', (25 % a**2) > b or (25 % b) < a)
```

```
5. False
```

```
print('6.', (25 % a**2) < b and (25 % b) > a)
```

```
6. True
```

```
print('7.', (12 / a) and (a * 4) < b)
```

```
7. True
```

Exercises

End of chapter Exercises

1. Write and execute a Python script to display your own name as an output in the terminal.
2. Write and execute a Python script that:

- Displays the text `Please press enter to continue...`, and waits for the user to press enter.
- Once the user pressed enter, the program should display `Welcome to my programme!` before it terminates.

3. We have an enzyme whose reaction velocity is $v = 50\,mol{\cdot}L^{-1}{\cdot}s^{-1}$ at the substrate concentration of $[S] = K_m = 2.5\,mol \cdot L^{-1}$. Work out the maximum reaction velocity or $V_{\max}$ for this enzyme using the Michaelis-Menten equation:

$$v = \frac{V_{\max}[\mathrm{S}]}{K_m + [\mathrm{S}]}$$

Q1

```
name = 'Gerold Baier'

# Displaying the author's name:

print(name)
```

```
Gerold Baier
```

Q2

```
# Blocking the execution until the user
# presses enter:

input('Please press enter to continue...')

```

```
6 print('Welcome to my programme!')
```

Q3

```
1 v = 50 #mol/L/s
2 k_m = S = 2.5 #mol/L
3
4 # Rearranged the equation to
5 # solve for v_max:
6
7 v_max = (v * (k_m + S)) / S
8 # Unit: mol/L/s
9 print('Vmax =', v_max, '[ mol / (l * sec) ]')
```

```
1 Vmax = 100.0 [ mol / (l * sec) ]
```

Keypoints

- Two key functions for I/O operations are print() and input()
- Three most commonly used variables such as **`int`**, **`float`**, and `str`.
- Variable scope can be local or global depending where they are being used.
- Mathematical operations follow conventional rules of precedence
- Logical operations provide results in Boolean (True or False)