Python Fundamentals 1

Input/Output Operations PF1

Learning Objectives

1 Understand how information can be inputted and outputted in Python
2 The basics of variables and data types
3 What are strings and how to use them
4 Basic mathematical operations and how to use them

Introduction

Before we can perform any analysis, build any models or even structure a program, we need to master some of the mos basic tools Python provides: the ability to **send information out** (output), **receive information in** (input), and **store and work with that information** using **variables** and **data types**.

This Python Fundamentals lesson walks through these core concepts.

KEY TERMS

Variables:

Data Types:

Input and Output (I/O) Operations

What are I/O operations?

In computer science, **input/output (I/O)** refers to the communication between a computer program and the outside world. This might mean displaying the result of a computation on the screen, reading data from a file, prompting a user to enter their name, or receiving input from a device like a microscope, sequencer or camera. These interactions are crucial in scientific programming, where we often need to load datasets, prompt for experiment parameters, and print results.

Python provides two simple but powerful built-in functions to handle basic I/O:

- `print()` -- used to send output to the screen
- `input()` -- used to receive user input

These two functions make it possible to build interactive programs, pipelines and command-line tools.

Video unavailable:

No URL provided

What is a function in Python?

Before we delve further, let's get to grips with what a **function** is, and what it does, in Python. A function is a **named block of code that carries out a specific task** whenever you ask it to. We have just introduced two of the most commonly used functions: `print()` and `input()`. Both are built-in functions, which means they are provided by Pytho itself, and are not written or defined by the user, or imported from a specific module or package; they are a mainstay of

your plain Python installation. We will explore modules and packages further in **PF3**.

Although they are both functions, `print()` and `input()` do actually behave quite differently to the majority of other functions. For instance, the `print()` function takes information you give it and sends it outward to the screen. By contrast, the `input()` function waits for information to come in from the user and then returns that text back into your program. This illustrates two important points about functions:

- Some functions act mainly to produce an effect (like displaying text on the screen).
- Others are designed to produce a value that can be stored and used in further calculations.

This is the surface-level definition of what Python functions are, and is all we need to know, at this point in the course. The specifics of function writing and defining are covered in later lessons, but the remainder of this lesson will now focus on `print()` and `input()`.

Print

The `print()` function:

As described above, the `print()` function is used to send textual output to the console or notebook. This also uniquely makes it the programmer's primary and most universal tool for **seeing what's happening** inside a script: whether that's displaying simple alphanumeric characters as a message, displaying the result of a calculation or operation, or even reporting errors. As we progress, we can even use `print()` to display the contents of data structures, such as variables, lists or dictionaries.

Syntax:

In order to make use of a function such as `print()` - or any other function for that matter - Python follows specific **syntax**. In this context, **syntax** refers to a set of rules that allow the **Python interpreter** (the part of your computer tha is decoding what your Python program is trying to say and do) to translate the letters, numbers, spaces, indentations an punctuation in your Python program, into a **valid instruction**.

Python has broad syntactic principles that are followed by many different functions but, furthermore, each specific Python function will have syntax of its own that helps the user very specifically control how it works.

Before we dive into the various syntax of the `print()` function, run the code cell below. Its output will display beneath it, on your console (or screen).

Printing a string:

```
1 print("Welcome to L2D.")
```

```
Welcome to L2D.
```

Note the Python function itself is called by typing the word `print`. It is then fed **arguments** in round parentheses. An **argument** is a piece of information that you pass into a function so that the function knows what to work with. Think o a function like a machine: the arguments are the *items you feed into the machine*, and the function then *processes* them

in some way to give an output. Without arguments, many functions would not know what you want them to do, as they would have nothing to act on or process.

If we revert to the `print` statement in the code cell above, we can display exactly what we are writing: and to do this, we must specify the data as a **string**. A **string** is essentially a sequence of alphanumeric characters, that `print` can display in exactly the order in which they are typed. These will be covered in considerably more depth in our Python Fundamentals 2 lesson.

Specifying your argument as a string is achieved by using either **single** (`'...'`) or **double** (`"..."`) **quotation marks**. The use of single or double quotes is *interchangeable*, which also allows you to display quotes inside the output if you wish.

For example:

```
1 print('Welcome to L2D!')   # single quotes
2 print("Welcome to L2D!")   # double quotes
```

Welcome to L2D! Welcome to L2D!

PRACICE EXERCISE

Have a pratise in the cell below printing whatever you want to the screen:

- The cell or block below actually runs the Python in your browser

```
# Type your print
statement here and
```

Run Code

Loading Pyodide...

Comments:

Note also the # symbol, next to or above each `print` statement. In Python, everything that follows # is called a **comment**; that is, it is **excluded** from processing. This is a very useful feature that we can use to annotate our code in any way that we like, without having it be picked up and executed by the Python interpreter. In this case, this has allowed us to label the line with strings specified by double quotes, and single quotes, respectively.

As aforementioned, the interchangeability of single and double quotes provides the advantage of being able to display single or double quotes as part of the string itself - or as part of the displayed output, as follows:

```
1 print('A student said: "L2D is a superb course."')
```

```
```

A student said: "L2D is a superb course."

In this example, we want the speech marks (double quotes) to appear in the printed output; hence, we have made use o single quotes to define the string. However, if we used double quotes around the whole sentence to specify the string to print (which already contains double quotes), we would end up with two pairs of double quotes next to each other, which would produce an error:

```
# Run this cell to
see the error
```

Run Code

Loading Pyodide...

Therefore, interchangeability of double and single quotes when specifying a string in Python, provides a neat solution t this problem.

NOTE

While it does not matter if you use single or double quotation marks in your code, it is advised good practice to stick to one, and use this throughout your code, for the purposes of consistency and legibility. Of course, if you need to change over to one or the other, and this cannot be avoided because your string contains a single or double quote that you want to display, then it is acceptable to use both. But, in general, we advise picking one, and sticking to it.

Optional arguments:

Aside from printing strings, `print` is capable of a good deal more. To explain this, it is important to understand a furthe two types of argument in Python: **optional** and **keyword arguments**.

To understand this further, let's try and print multiple strings, in sequence. To do this, we pass each string as an argument into `print`, and we separate the strings out from one another using a comma: `,`

```
1 print("We are", 'programming in Python.')
```

We are programming in Python.

In this example, notice that there are no spaces after the word 'are' or before the word 'programming'. There are two strings defined here, and yet somehow, `print` has automatically inserted a space between these two words, in the resulting output.

The reason for this, is that the print function has a **default** separator between items fed into it as arguments, and this is simple **space**. This default setting, however, can be modulated and changed. For instance. We could specify that '---' is

the separator that is used. In order to modulate this default setting we need to use what we term a **keyword argument**, and specify the value that we want it to replace the default of a space with. In the case of the `print` function this keyword argument is `sep`. By default, the value that `sep` is set to is a space: `sep=' '`, as indicated by the single space between the two quotes.

The `sep` keyword argument is also termed an **optional argument** because, when it isn't specified by the user, it has a **value set to it, by default**. As we go onto explore more functions in Python, you will see some keyword arguments arise that don't have default values, and are 'uncalled' by default, until specified by the user. But for now, `sep` has a default value, and is thus termed an optional argument.

To prove this point, let's take the exact same line, and add in the optional argument `sep`, and set its value to be '---'.

```
1 print("We are", 'programming in Python.', sep='---')
```

```
We are---programming in Python.
```

Now you can see clearly, that the separator between arguments is set to the `---` (three hyphens) that we specified.

Escape sequences:

The `print()` function will also respond to **escape sequences**, which are character combinations that always begin with `\`. They indicate display actions or characters that are not possible to type plainly into a string: for instance a 4-space ta indentation, or a new line.

The most commonly encountered escape sequence is probably the newline character `\n`, which tells print to display everything that follows it, on a new line.

Other common escape sequences, their apparent results, and appropriate use-cases are listed in the table, below.

Escape Character	Description	With Escape Character	Output	When to Use
`\n`	Newline (line break)	`Welcome\nto L2D!`	`Welcome` `to L2D!`	Creating multi-line output, formatting text blocks, separating lines
`\t`	Tab character	`Welcome\tto L2D!`	`Welcome to L2D!`	Creating columns, indenting text, aligning dat in output
`\"`	Double quote	`Welcome\"to L2D!`	`Welcome"to L2D!`	Including literal quotes inside double-quoted strings: `"She said \"Hi\""`
`\'`	Single quote	`Welcome\'to L2D!`	`Welcome'to L2D!`	Including apostrophes inside single-quoted strings: `'It\'s working'`
`\\`	Literal backslash	`Welcome\\to L2D!`	`Welcome\to L2D!`	File paths on Windows, regex patterns, when you need an actual backslash
`\r`	Carriage return	`Welcome\rto L2D!`	`to L2D!` (overwrites)	Progress bars, overwriting text on same line, Windows line endings
`\b`	Backspace	`Welcome\bto L2D!`	`Welcomto L2D!`	Removing characters, creating special text effects (rarely used)

Escape Character	Description	With Escape Character	Output	When to Use
`\0`	Null character	`Welcome\0to L2D!`	`Welcome` (null byte)	Binary data, C-style string termination (advanced use)
`\f`	Form feed	`Welcome\fto L2D!`	`Welcome` `  to L2D!`	Page breaks in printing, legacy formatting (rarely used)
`\v`	Vertical tab	`Welcome\vto L2D!`	`Welcome` `  to L2D!`	Vertical alignment in old systems (rarely used

Note: The actual visual output may vary depending on your terminal or display environment, especially for `\r`, `\b`, `\f`, and `\v`.

Another commonly encountered optional argument is `end`. This controls what is displayed at the end of the output. For instance, if you use two print statements in sequence, in a single block of code, the `end` optional argument is set to a default of `\n`. This would print the first statement, and after it, a newline character; so that the second print statement, begins on the line after. This is demonstrated in the cell, below:

```
1 print('Welcome to L2D!')
2 print('A biosciences-oriented Python course.')
```

Welcome to L2D! A biosciences-oriented Python course.

However, if we use the `end` optional argument, and set this value to a a single space `' '`, notice that the second print statement outputs on the same line:

```
1 print('Welcome to L2D!', end=' ')
2 print('A biosciences-oriented Python course.')
```

Welcome to L2D! A biosciences-oriented Python course.
PRACTICE EXERCISE

1.

In the cell below, feel free to play around with different values for `sep` and `end`, and get a feel for using print. The code in the cell below is editable so, once you have made your edits, run the code, and the output will reflect your changes. I is recommended to try a few different values, combinations *etc* to test out the principles introduced up to this point in the lesson.

```
# Example 1:
print('In this cell',
```

Run Code

Loading Pyodide...

Other optional arguments for `print()`:

In addition to `sep` and `end`, the `print()` function has other keyword arguments that give you more control. These include the following, and are given in brief overview, as we won't use them too often in this course:

- The `file` argument tells Python where to send the output — by default this is the console (`sys.stdout`), but you can redirect it to a text file or another stream.
- The `flush` argument is a setting that forces the output to be written immediately rather than being temporarily stored in a buffer (this can be important in real-time logging).
- Less commonly used but still available are `encoding` and `errors`, which matter when writing text containing non-ASCII characters to a file or stream: `encoding` specifies how characters should be represented (for example `"utf-8"`), while `errors` controls what Python does if a character cannot be encoded (for instance `"ignore"`, `"replace"`, or `"strict"`). In everyday use you may never need to set these, but they are powerful options when working with files, international text or streams.

Variables: Storing Data

A variable is a label that refers to data stored in memory. Think of it as a simple name that we give to an empty container (imagine it as an empty box) into which we can store data. You can store multiple different data types in a variable: anything from an integer, a string - or even other data structures such as lists or dictionaries (covered in **PF2/3**).

Variables allow you to assign, update, reuse, and manipulate values. They allow you to use a simple name that can occur multiple times in a block of code. You can keep this variable name the same, and repeatedly alter the data stored within it, allowing a single change to affect every downstream occurrence of that variable in your code. This simple bu powerful feature makes managing larger scripts much easier. Variables are vital to meaningful coding in Python, and can help manage your code and that data you process with it, in a neat and efficient way.

Creating a variable, and storing data in it:

In Python, we create a variable by using the **assignment operator** `=`.

NOTE

It is important to note `=` symbol in Python does *not* mean "equals" in the mathematical sense. instead, it means *assign the value on the right-hand side to the name on the left-hand side*.

The process of assigning data into a variable defines that variable, and is termed **assignment**.

The code cell below demonstrates the creation of two new variables (`answer` and `greeting`), with integer and string data being assigned to each, respectively. A variable can be named anything, contain alphanumeric and even punctuation characters, but cannot contain spaces. It is good practice to name a variable something logical or intelligible, pertaining to the data it contains.

In the example below, the variable names `answer` and `greeting` point to the integer **42** and the phrase "**Hello, L2D!**", respectively. When you use these variable names later, Python will look up the stored value.

You can also reassign variables to hold new values, and Python will simply update what the name points to:

```
1 # Assign the number 42 to a variable called answer
```

```
answer = 42

# Assign a string to a variable called greeting
greeting = "Hello, L2D!"

# Now we can use the variables in our program
print(answer)      # displays 42
print(greeting)    # displays Hello, L2D!
```

42 Hello, L2D!

One of the most useful features of variables, is that you can **reassign** data to them, and replace their contents with new contents, by simply calling another assignment, subsequent to the first. The last assignment line in your code, reflects the value stored in the variable, after that point. When a value is reassigned to a variable, Python simply updates (and replaces) its contents with the new value:

```
answer = 100
print(answer)    # now displays 100
```

100

Variables can store many types of data — numbers, strings, lists, dictionaries, and more. As your the programs you write develop and grow, variables let you keep track of data, perform calculations, and manage information without having to repeat yourself.
PRACTICE EXERCISE

2.

In the box below, try changing the value assigned to the variable `colour` and run the cell. Each time, changing only tha line, will automatically update the contents of the variable `colour`, in all of its subsequent instances, in the code.

- Try adding your own print statement to a line below

```
# Edit the string for
the variable colour
```

Run Code

Loading Pyodide...

Interactive Python

How it behaves:

One (traditional) way to run your Python code, is by using Python scripts, and running these from the terminal. This is where your whole script or block of code can be saved as a file with the extension '.py'; to execute the code within you .py Python script, it must be called and run directly from the terminal, like this:

```
python my_python_script.py
```

In order to create a .py file, you can do so in a simple text editor application, making sure to simply change the file extension to .py. Or you can use a specialised code editor or Integrated Developer Environment (IDE).

However, another way to run your Python code - similar to how we are doing in this page - is to run individual cells within a **Jupyter Notebook** - a commonly-used file format that supports individual cells of different formats (for instance - you can merge text and images in **markdown** cells, and have executable **Python** cells for storing and runnin code).

One distinct behaviour of a Jupyter Notebook is that you don't always have to explicitly use print. In a **Jupyter Notebook**, the output appears beneath the code cell. In a **Python script**, it appears in the terminal or shell window. Notably, Jupyter will also display the final value in a cell without needing `print()` -- a behaviour that doesn't occur in `.py` script:

```
x = "This is the contents of variable 'x'."

x  # iPython automatically displays the contents of variable x, just by leaving the variable
```

This is the contents of variable 'x'.

```
# Have a play with
this one
```

Run Code

Loading Pyodide...

F-strings

Variables can also integrate directly into print statements within a **formatted string literal**, also known as **f-strings**. F-strings offer a legible, flexible way to include variables or expressions directly into strings (the characters entered between single or double quotes).

Up until this point, we have been printing variables as individual, standalone arguments, separated from strings by a comma. The code cell below demonstrates how a variable can be directly incorporated into an f-string. The syntax of a f-string is that - as with an ordinary string - its start and end are contained within single or double quote marks, except these are preceeded by the letter `f`, with variables inserted directly into the strings in curly brackets, as below:

```
dose = 7.5

print(f"The dose is: {dose} mg.")
```

```
```

The dose is: 7.5 mg.

F-strings are not limited to simple text or numbers: you can use them with variables of many different **data types** in Python. These include integers (`int`), floating point numbers (`float`), strings (`str`), Boolean values (`bool`), as well as collections such as lists (`list`), tuples (`tuple`), and dictionaries (`dict`): we have not encountered many of these data types yet, but will introduce them more explicitly, later in the lesson.

When you insert a variable into an f-string, Python automatically converts its value into a string representation so that i can be displayed. You can even combine this with the `type()` function - a function that tells you what type of data an object contains - inside the curly brackets to see both the value of the variable and its data type. This makes f-strings a powerful and flexible way to check, debug, and present a wide variety of information in your programs.

```
# Examples of different data types

patient_id = 12345          # int
temperature = 36.7          # float
sample_name = "SMP001"      # str
is_positive = True          # bool

print(f"Patient ID: {patient_id} (type: {type(patient_id)}).\n")
print(f"Temperature: {temperature} (type: {type(temperature)}).\n")
print(f"Sample: {sample_name} (type: {type(sample_name)}).\n")
print(f"Test positive: {is_positive} (type: {type(is_positive)}).\n")
```

Patient ID: 12345 (type: <class 'int'>). Temperature: 36.7 (type: <class 'float'>). Sample: SMP001 (type: <class 'str'>). Test positive: True (type: <class 'bool'>).

You will encounter both f-strings and the `type()` function throughout this course.

The input() function

Converse to the `print()` function, which is used to send information **out** of a Python script and onto the console, the `input()` function is used to bring information **in**. When used, you can display a message on the console to the user, an allow them to type input in using their keyboard and - upon hitting return or enter - returns the collected input as a **string**.

NOTE

In Jupyter notebooks, `input()` creates an interactive text box below the cell. This is where you can type your input, an hit enter to complete the function's execution.

Run the cell below, type your name into the box, and hit enter.

```
# To work in a
browser this will
```

Run Code

Loading Pyodide...

Prompt customisation with `input()`:

Uniquely, whatever data type you enter into the `input()` function, it will **always convert this into a string**.

Run the code cell below, enter your age as an integer, and notice the data type that it returns for your collected input:

```
age = input("Enter
your age: ")
```

Run Code

Loading Pyodide...

In this example, the variable `age` now contains your age (which you entered as a whole number, or integer), stored as data of type string (or `str`). So far, we have encountered two data types in this lesson: string and integer (data type `str` and `int`, respectively).

If we then wish to convert this string data into an integer (data type `int`), we must explicitly convert data type `str` to `int`. We call this data conversion **typecasting**, which is covered in greater detail in the section on **data types**, later in this lesson.

In order to explicitly convert or typecast `str` into `int` when a user enters a whole number, you can **wrap** the `input()` function call with the `int()` function to convert the string into an integer. In programming, the term **wrapping** refers to placing one function call or operation *around* another; this is often done by placing one function call **inside the round parentheses of another**. In this way, **the output of the inner function, becomes the input of the outer function**.

Run the code in the cell below, and you will notice that the input is collected and typecasted, in a single line. The contents of the `age` variable, is now the user-collected input given as an integer (data type `int`).

```
age =
int(input("Please
```

Run Code

Loading Pyodide...

PRACTICE EXERCISE

3.

Create an input that asks the user to enter their home town and save it to a variable:

- Format an appropriate print statement for the given answer

Data types in Python

Every value in Python has a **type**. We have looked at types `str` and `int`, so far, and have introduced the `type()` function, as a useful way of determining what type of data is stored in a variable.

The importance of types in Python include informing the programmer as to what kind of information they are dealing with, and this subsequently defines what can sort of operations can be performed on it.

For instance, `"42"` is a string type (text), while `42` is an integer type (whole number). Even though they look similar, Python treats them very differently: you can use logical operations to calculate `42 + 1` to obtain `43`. However, doing so with a string simply concatenates (or joins) them, with `"42" + "1"` giving `"421"`. In short, types are **like labels** that tell Python what kind of information you're working with, ensuring that its operations behave, predictably.

In more methodical summary, types:

- **Control behaviour** — the same operator or function can act differently depending on the type.
- **Prevent errors** — Python will stop you from adding a number to a string, because it wouldn't make sense.
- **Guide your thinking** — knowing the type of your data helps you decide what methods or functions are availabl
- **Support flexibility** — Python has many built-in types (strings, integers, floats, lists, dictionaries, etc.), and you can also define your own.

Built-in Types in Python:

Below is a table of the various data types built into Python.

Type	Example	Description
`int`	`42`	Whole numbers
`float`	`3.14`	Decimal numbers
`str`	`'geneX'`	Text data
`bool`	`True`, `False`	Boolean values for logical operations
`complex`	`2 + 3j`	Complex numbers
`list`	`[1, 2, 3]`	Ordered, mutable sequence
`tuple`	`(1, 2)`	Ordered, immutable sequence
`dict`	`{'a': 1}`	Key-value pair mapping
`set`	`{1, 2, 3}`	Unordered collection of unique items
`NoneType`	`None`	Represents null or absence of value

Across this course, and with many biological and medical datasets, you will likely encounter almost all of the types listed, above.

Type conversion and typecasting:

Python allows **type conversion**, also known as **typecasting**, to change a value's type. This is crucial when dealing with data from external sources, user input, or read-in files.

Typecasting is often necessary when the type of data you have is not the type you need for an operation. Our earlier example of the `input()` function above, explains why it could be necessary.

If you want to perform calculations with that number, you must *typecast* the string into an integer using the `int()` function, or into a floating-point (decimal) number using the `float()` function. Likewise, you can turn numbers into strings with the `str()` function, which can useful for combining numbers with text, to give customised output.

Aside from `str` and `int`, typecasting can also be important when moving between other types of data. For instance, typecasting can also come up when working with numeric data, booleans (`True` or `False`), and missing values.

As an example, you might be analysing gene expression levels, measured as floating-point (decimal) values. You may want to simplify these into a simpler expressed *vs.* not expressed binary, and one way to do this is by typecasting the values into a **Boolean** value of data type `bool`.

A **Boolean** is a data type in Python that represents one of two possible values: `True` or `False`. Booleans are used to express logical conditions — for example, whether a statement is correct, whether a comparison holds, or whether something is present or absent.

You can convert a data calue into a Boolean using Python's built-in `bool()` function. This converts a value into a Boolean (`True` or `False`), following these two simple rules:

- Values that are **non-empty** or **non-zero** become `True`.
- Values that are considered **empty** or **zero** become `False`. This also includes the datatype `None` given in the table above for values that are missing, which are also converted to the Boolean `False`.

This is an implicit condition of this function. Later in this lesson we will explore **conditional statements** - a construct in Python used to execute a block of code depending on whether a specific condition is met, or not. In a nearly similar fasion, the `bool` function implicitly *evaluates* a value: with non-zero being converted into `True`, and zero or missing values are `False`.

In the cell below, try changing the value assigned to the variable `expression_level` and see how the `bool` function responds:

```
expression_level =
1.0 # Gene expression
```

Run Code

Loading Pyodide...

TIP

As shown above, a zero/empty value becomes `False` and non-zero/empty become `True` with `bool()`. However, the values `False` and `True` are also equivalent to `0` and `1` themselves. You can actually substitute boolean values for the integers `0` and `1` if you wanted to. Although it is more readable to use the text based versions.

Mathematical Operations

Python also has built-in syntax that is very useful for performing simple calculations, without the need to import or install any external libraries or modules. The syntax for these are listed, below.

Operator	Name	Description
`+`	Addition	Adds two values together
`-`	Subtraction	Subtracts the right value from the left value
`*`	Multiplication	Multiplies two values together
`/`	Division	Divides the left value by the right value (always returns data type `float`)
`//`	Floor division	Divides and rounds the result down to the nearest whole number (always returns data type `int`)
`%`	Modulus (remainder)	Returns the remainder after division
`**`	Exponentiation	Raises the left value to the power of the right value

Observe the output of the code in the cells below, to see each of these operators in action:

```
# Let's begin by defining two variables to use in our calculations:

a = 12
b = 5
```

```
# Addition:

a + b
```

17

```
# Subtraction:

a - b
```

7

```
# Multiplication:

a * b
```

60

```
# Division:

a / b
```

2.4

```
# Floor division:
```

Loading Pyodide...

```
# Modulus
(remainder):
```

Loading Pyodide...

```
# Exponentiation:
```

Loading Pyodide...
PRACTICE EXERCISE

4.

Have a play around with the different mathematical operations in the code cell below:

```
# You can edit these
variables
```

Run Code
Loading Pyodide...

Absolute value

The `abs()` function:

Alongside operators, Python also provides another useful built-in function for working with numbers. This is the `abs()` function, which returns the **absolute value** of a number. The absolute value represents the number's distance from zero with any **negative sign removed**.

Run the code in the cell below to understand how this works:

```
# Absolute difference
in cell count
```

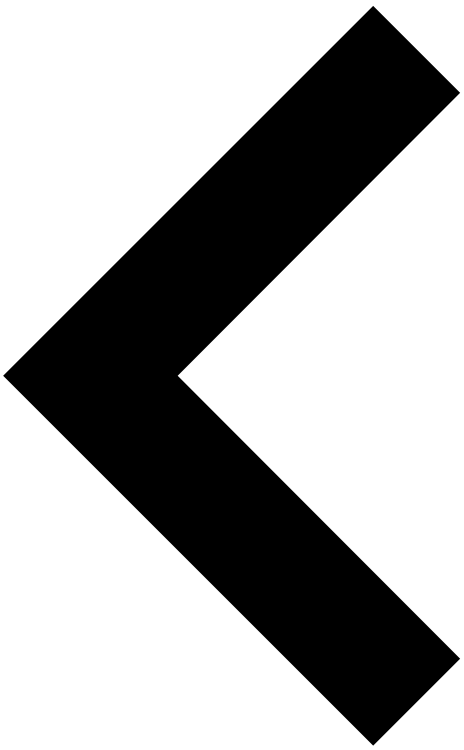
Run Code
Loading Pyodide...

In this example, you can see how `abs()` is particularly useful when dealing with changes, **where only the magnitude of the difference matters**, irrespective of whether the difference represents an *increase* or *decrease*.

In the example code cell, above, a drop in white blood cell count of `-1200` and a rise of `1200` both have the same absolute change in value. And in such situations, using `abs()` can be a useful way of standardising lots of output.

Summary

In this lesson, we introduced several core concepts in Python that form the foundation of all programming in the language. We learned that **input and output (I/O)** are handled using `print()` for displaying information and `input()` for receiving user input. We introduced **functions** as named blocks of code that perform specific tasks, able to accept arguments inside parentheses. We explored **strings** as sequences of characters that can be manipulated with **escape sequences** and formatted dynamically using f-strings. Code annotation and instruction was introduced in the form of in line **comments**, landmarked by `#`, allowing us to document our code without affecting how it runs. We also looked at **variables**, which let us store, reuse and modify values, and at **data types** such as `str`, `int`, `float`, and `bool`, which define the kind of information being processed. **Typecasting** enables conversion between these data types when needed while **Booleans** (`True` and `False`) underpin logical operations, with Python automatically interpreting values as true and false, respectively. Finally, we examined Python's **mathematical operators** (`+`, `-`, , `/`, `//`, `%`, `*`) for their use in calculations, together with the `abs()` function, which returns the absolute value of a number: useful when only the size of a change matters, irrespective of a number's polarity.



Previous Next