



TASK

Getting Started with Your Bootcamp

Visit our website

WELCOME TO THE TASK ON GETTING STARTED WITH YOUR BOOTCAMP!

Well done on making one of the most important decisions of your life. Moving into a tech career can at times feel alienating, but we are here to walk with you throughout your programming journey with us.

To kick-start this journey, we are going to introduce you to the command line by using package managers, setting up your development environment, and computer programming. We are going to introduce you to a workflow that will not only serve you throughout this bootcamp, but for the rest of your life.

The command line and terminal

As a software engineer and data scientist, it is essential that you can use libraries and frameworks to be able to build applications. However, to do this, it becomes important to familiarise yourself with the command line. The command line is a tool that you will use often as a software engineer, and you will use it for many subsequent tasks.

WHAT IS THE COMMAND LINE AND WHY DO YOU NEED IT?

The command line is a means of interacting with a computer program where the user issues commands to the program in the form of successive lines of text. With the command line, you can quickly issue instructions to your computer, getting it to do precisely what you want it to do. The command line is rarely used by most end users since the advent of the graphical user interface (a more visual way of interacting with a computer, using items such as windows, icons, menus, etc.).

For software engineering and data science, you will find it helpful to use the command line when interacting with your files, especially those created using frameworks and libraries. You will also need to be familiar with the command line to work with version control systems like Git. Hence, this task will allow you to acquaint yourself with some of the basics of the command line.

FINDING THE COMMAND LINE



In Windows, you can simply click the Start menu and type **cmd** in the search box to locate the command line. Alternatively, the command line should be one of the options under 'Programs' and you can simply click on the application to open it.



With macOS, open the command line by opening the terminal. This can be done by opening the Applications folder, navigating to Utilities, and then launching Terminal. Alternatively, you can search for "terminal" to find the application to launch.

COMMON COMMANDS

The following table provides a selection of commonly used PowerShell and Unix commands to help get you started with the command line.

For a comprehensive list of commands, visit these pages for [PowerShell \(Windows\) commands](#), [Command Prompt \(Windows\) commands](#), and [Unix \(macOS/Linux\) commands](#) to explore more command line operations.

Description	Windows cmd	Windows PowerShell (alias)	macOS/Linux
Displays the current working directory	chdir	pwd	pwd
Changes the directory	cd	cd	cd
Move up one level in the directory	cd ..	cd ..	cd ..
Displays a list of a directory's files and subfolders	dir	dir	ls
Print contents of a text file	type	type	cat
Create a new directory	mkdir	mkdir	mkdir
Remove files and directories	del / rmdir	del / rmdir	rm
Move or rename files and directories	move / ren / rd	move / ren	mv
Copy files and directories	copy	copy	cp
Clear the screen	cls	cls	clear

Quit the terminal	<code>exit</code>	<code>exit</code>	<code>exit</code>
-------------------	-------------------	-------------------	-------------------


You can get detailed information, including instructions, examples, and usage guidelines, on the various commands available in both Windows PowerShell and Unix-based systems.

For **Windows PowerShell**, open the terminal and run **Get-Help**. To get help on a specific command, run **Get-Help <cmd-name>**.

For **Windows Command Prompt**, open the terminal and run **help**. To get help on a specific command, run **help <cmd-name>**.

For **Unix-based systems**, open the terminal and run **man** to access the manual pages. To get help on a specific command, run **man <cmd-name>**. Additionally, you can also run the **whatIs <cmd-name>** to get a brief description of the specified command.

As you can see, the command line has the built-in **help** (Windows) or **man** (macOS/Linux) command. This can be used to view all the commands that are executable. At this point, why not type the **help/man** command into the command line of your computer and hit Enter to find out more about all the commands? To get help on a specific command, you have to type **help** followed by the command in Windows like so:



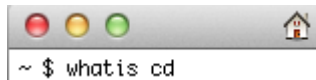
```
C:\Windows\system32\cmd.exe
C:\Users\User>help cd
```

Or type **man** followed by the command in macOS/Linux:

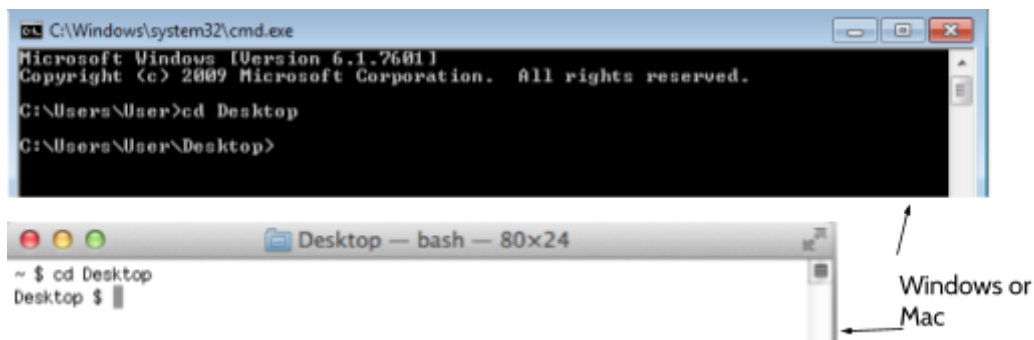


```
~ $ man cd
```

You could also type **whatIs** followed by the command in macOS/Linux to get help. Compare the output you get with the **whatIs** command with the output from the **man** command:

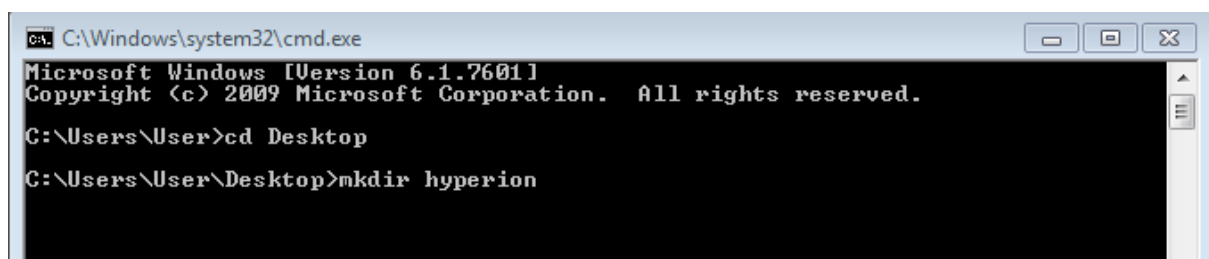


The command (in the images above) will give you the information about the **cd** command. As will be noted by the information provided by the command line, the **cd** command is used for navigation. It takes you from one directory to the next. For example, if you want to perform some command on a folder that is on your desktop, you would have to type **cd** to change the directory to your desktop as shown in the image below:



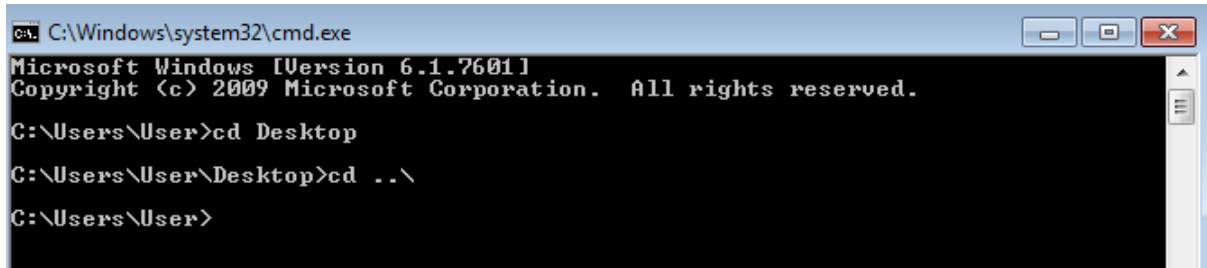
From here, we can now perform operations on the files or folders in our desktop, since we have navigated into it. But, what if we have forgotten the name of the file or folder that we wanted to operate on? Well, you can simply use the **dir** (Windows) or **ls** (macOS/Linux) command to get a list of all the files or folders saved on the desktop.

But let's not alter any file or folder on the desktop; instead, let's create a new folder. Do you recall the command to make a new folder? That's right, it's **mkdir**.



Notice that we have made a new folder on the desktop called 'hyperion'. It's that simple! So, now that we have done what we wanted to do on our desktop, how do we get back to where we were, i.e., how do we navigate **backwards**?

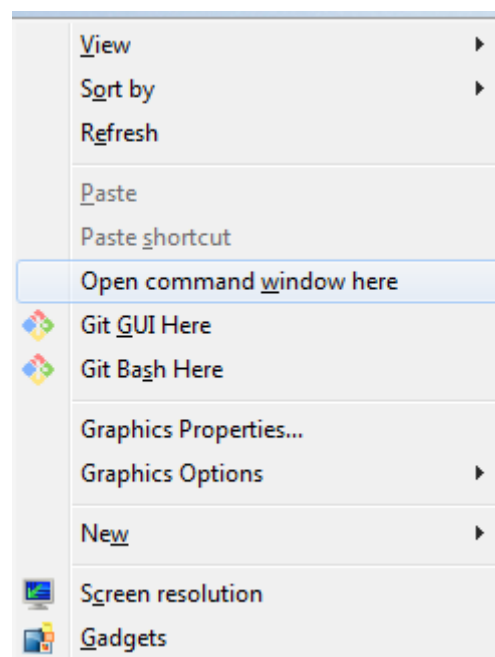
To navigate two directories back, we would have to type **cd ../../**. However, navigating back and forth may seem tedious to do.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\User>cd Desktop
C:\Users\User\Desktop>cd ..\
C:\Users\User>
```

Wouldn't it be nice if we could figure out a way in which we could open a command window in any directory with minimal effort? Fortunately, you can with Windows! Simply hold shift and right-click on a folder or empty space to open a command window in that directory:



Your computer may display 'Open PowerShell window here'. PowerShell is similar to the command window and will accept most of the same commands.

SCRIPT FILES

As you advance in your skills as a software engineer, you may at times find that there are certain commands that you use repeatedly. Instead of retyping these commands into the command line every time, you can create a script file that contains these sets of commands that can be executed as needed. Often such files will be executed periodically, e.g., daily, weekly, monthly etc. In Windows, we can create batch files and in Mac and Linux systems, we create shell scripts.

Batch files

A batch file is a script file in DOS, OS/2, and Windows. Batch files are normally used by individuals who run the same commands frequently. Instead of typing out the commands each time, the commands are simply placed in a batch file. To execute the commands contained within a batch file, you can simply double-click it.

The batch file consists of a series of commands to be executed by the command line, stored in a plain text file. To create a batch file, you have to open a plain text editor (e.g., Notepad) and navigate to File > Save As, and in the 'Save As' window, input the name for your batch file and then add a '.bat' extension, e.g., *mybatch.bat*.

Bash (shell) scripts

To create a shell script:

1. Open a text editor (e.g., gedit or TextEdit).
2. Add the following instruction: **#!/bin/bash** to the first line of the script file.
3. On the following lines enter the instructions that you would usually type into the terminal, one line per instruction.
4. Save the file. It is not a requirement but it is common practice to save your file with a .sh extension. To save the file properly you may need to specify that the file is a plain text file. Do this by selecting Format > Make plain text.
5. Make this file executable by typing the following into the command line: **chmod +x myscript.sh** where **myscript.sh** is the name of the script file.
6. To run the script type: **sh myscript.sh** where **myscript.sh** is the name of the script file.

SPOT CHECK

Let's see what you can remember from this section.

1. What are the two benefits of using the command line as a software engineer or data scientist?
2. What is the command to display the name of or change the current directory?
3. What is the command to display a list of files and subdirectories in a directory?

PACKAGE MANAGERS

Package managers provide a means to install packages, organise them in your file system, and manage dependencies between packages. A dependency is when a package depends on another particular version of another package to function. If you have two different packages that rely on different versions of another package, this creates problems if you don't have a package manager. It can also be very time-consuming to find and install all the dependencies for a package. Package managers also check for known vulnerabilities that may pose a security risk. You can use a package manager to share packages you have created with others.

There are two types of package managers: those for **operating systems** and those for **programming languages**. Package managers are linked to software repositories where the packages or software are stored.

Some common operating system package managers include:

- **Chocolatey** for Windows ([Chocolatey repository](#))
- **Homebrew** for macOS ([Homebrew repository](#))
- Linux – each distribution has its own package manager:
 - **DNF** for Fedora
 - **APT** for Ubuntu

As mentioned, programming languages typically have language-specific package managers. Examples of some programming languages and commonly used package managers are summarised in the table below:

Language	Package Manager	Software Repository
Python	<code>pip</code>	PyPI (Python Package Index)

Python	conda	<u>Anaconda Packages</u>
Java	Maven	<u>Maven Central</u>
Java	Gradle	<u>Maven Central</u>
JavaScript	npm	<u>Node Package Manager</u>
Javascript	yarn	<u>Yarn Registry</u>

Python package managers

pip is the general-purpose package manager for Python. It is used to install, upgrade, and manage Python packages from the PyPI repository. On the other hand, **conda** is more specialised towards data science programming, providing easy installation of both Python packages and other data science tools. This makes it an ideal choice for data-related tasks, scientific computing, machine learning, and other data-intensive projects.

SPOT CHECK ANSWERS

1. For software engineers, it is helpful to use the command line when interacting with files, especially those created using libraries and frameworks, such as Django. It also allows you to work with version control systems like Git.
2. **cd**
3. **ls** (macOS/Unix) or **dir** (Windows)

CONTAINERISATION

As you start working on software applications and larger projects that need to be deployed, you will also need to consider how to bundle your code with all the third-party packages it relies on.

For Python projects, the **venv** module is a well-known containerisation method. The [venv module](#) allows you to create isolated virtual environments for Python projects, each with its own Python interpreter and package dependencies. Additionally, Anaconda allows you to create specialised [conda virtual environments](#), which is particularly useful for data science projects, to package and manage your projects efficiently.

Installation

This guide will help you locate and use an installation script to set up your coding environment and also provides instructions for what to do in rare cases where the installation script fails. If you would like to install additional third-party packages, you should be aware of best practices relating to installation. The use of a package manager is highly recommended, and containerisation is important when different projects or applications require specific requirements.

SETTING UP YOUR DEVELOPMENT ENVIRONMENT

Please systematically and carefully follow the steps just below. Remember to read the following instructions in full before you implement them. Please note that these instructions are only valid for macOS, Windows, and Linux versions supported by their respective developers. If your operating system is not supported due to obsolescence, or if you're using an unsupported operating system such as ChromeOS, please ignore the instructions below and simply use <https://repl.it>, and when you need to work with Jupyter notebooks later in your bootcamp, <https://colab.google/>. Please also consider using ReplIt if you run into errors with the instructions below.

1. Visit <https://code.visualstudio.com/>
2. Download the version of VS Code that matches your operating system (OS). Alternatively, you can follow the instructions stated in the following links for the corresponding operating system families:
 - a. **macOS:** <https://code.visualstudio.com/docs/setup/mac>
 - b. **Linux:** <https://code.visualstudio.com/docs/setup/linux>
 - c. **Windows:** <https://code.visualstudio.com/docs/setup/windows>
3. For this step, we are going to install Python. Unix-like operating systems such as macOS and Linux often come with a pre-installed version of Python. It is generally discouraged to use the distributions of Python that are shipped with macOS as they may either be outdated or have customisations that might give you issues further down the line.

<https://code.visualstudio.com/docs/python/python-tutorial>.

- a. Ensure that if you are on Windows or macOS you have installed the latest stable version of Python using the prescribed means on the link above.

- b. Ensure that if you are on Linux and the prepackaged Python version is not the latest stable version, you get a package provider for your operating system that uses the latest stable version. Being behind by 1 or 2 minor versions is fine on operating systems such as Fedora. However, on operating systems such as Ubuntu, we strongly recommend that you use a PPA (Personal Package Archive) such as <https://launchpad.net/~deadsnakes/+archive/ubuntu/ppa>.
 - i. Please avoid the flatpak or snap versions as they can result in problems. Only proceed with flatpaks if you are sure of how they work.
 - c. For all operating systems, ensure that your environment paths are up-to-date with regard to your installation.
 - d. Per the guidelines linked above, ensure that you install the latest stable version of Microsoft's Python extension available from <https://marketplace.visualstudio.com/items?itemName=ms-python.python> so that you get tooltips and other useful tooling that help you as you program.
- 4. Use the compulsory task to ensure that your setup is working as expected.
 - 5. Use <https://code.visualstudio.com/docs/languages/python> to learn how to use Visual Studio Code with Python. If you've never programmed before, we strongly recommend you watch the videos.
 - 6. There are a range of other editors that you can use such as vi, emacs, Notepad++, and PyCharm, but we cannot guarantee that your peers will be familiar enough with them to assist you with them or that the academic staff members will be able to consistently review your work.
 - 7. If you're concerned about opt-out telemetry with Visual Studio Code, please turn it off by using the instructions from:
https://code.visualstudio.com/docs/getstarted/telemetry#_disable-telemetry-reporting

You can visit the [Python overview](#) page to learn how to use VS Code with Python. If you've never programmed before, we strongly recommend that you watch the [introductory videos](#). If you have a particular problem, [Stack Overflow](#) has several posts about VS Code.

Your first computer program and using variables

This task will teach you about creating your first computer program using variables by introducing the concept of pseudocode. Although pseudocode is not a formal programming language, it will ease your transition into the world of programming. Pseudocode makes creating programs easier because, as you may have guessed, programs can sometimes be complex and long, so preparation is key. It is difficult to find errors without understanding the complete flow of a program, and writing pseudocode helps you to establish this high-level understanding.

Next, you will be introduced to the Python programming language. Python is a widely used programming language, and it is consistently ranked in the top 10 most popular programming languages as measured by the [TIOBE programming community index](#). Many familiar organisations make use of Python, such as Wikipedia, Google, Yahoo!, NASA, and Reddit (which is written entirely in Python).

Python is a high-level programming language, along with other popular languages such as Java, C#, and Ruby. High-level programming languages are closer to human languages than machine code. They're called 'high level' as they are several steps removed from the actual code that runs on a computer's processor.

This task is a gentle introduction to Python, where you will be asked to create a simple program. In doing so, you will become familiar with the structure of a Python program.

Lastly, you will also be introduced to the concept of variables and the more complex programming problems that can be solved using these. A variable is a computer programming term that is used to refer to the storage locations for data in a program. Each variable has a name that can be used to refer to some stored information known as a value. By completing this task you will gain an understanding of variables and how to declare and assign values to them, as well as the different types of variables and how to convert between types.

INTRODUCTION TO PSEUDOCODE

What is pseudocode? And why do you need to know this? Well, being a programmer means that you will often have to visualise a problem and know how to implement the steps to solve a particular conundrum. This process is known as writing pseudocode.

Pseudocode is not actual code; instead, it is a detailed yet informal description of what a computer program or algorithm must do. It is intended for human reading rather than machine reading. Therefore, it is easier for people to understand than conventional programming language code. Pseudocode does not need to obey any specific syntax rules, unlike conventional programming languages. Hence it can be understood by any programmer, irrespective of the programming languages they're familiar with.

As a programmer, pseudocode will help you better understand how to implement an algorithm, especially if it is unfamiliar to you. You can then translate your pseudocode into your chosen programming language.

Pseudocode is easy to write and understand, even if you have no programming experience. You simply need to write down a logical breakdown of what you want your program to do. Therefore, it is a good tool to use in business to discuss, analyse, and agree on a program's design with a team of programmers, users, and clients before coding the solution.

WHAT IS AN ALGORITHM?

Simply put, an algorithm is a step-by-step method of solving a problem. To understand this better, it might help to consider an example that is not algorithmic. When you learned to multiply single-digit numbers, you probably memorised the multiplication table for each number (say n) all the way up to $n \times 10$. In effect, you memorised 100 specific solutions. That kind of knowledge is not algorithmic. But along the way, you probably recognised a pattern and made up a few tricks.

For example, to find the product of n and 9 (when n is less than 10), you can write $n - 1$ as the first digit and $10 - n$ as the second digit (e.g., $7 \times 9 = 63$). This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm! Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms.

One of the characteristics of algorithms is that they do not require any intelligence to execute. Once you have an algorithm, it's a mechanical process in which each step follows from the last according to a simple set of rules (like a recipe). However, breaking down a hard problem into precise, logical algorithmic processes to reach the desired solution is what requires intelligence or computational thinking.

Generally, an algorithm should usually have some input and, of course, some eventual output.

Now that you understand the concepts of pseudocode and algorithms, let's look at an example of how to write an algorithm in pseudocode that validates passwords (i.e., checks whether an entered password is correct).



Problem

Write an algorithm that asks a user to input a password, and then stores the password in a variable (something you will learn more about later in this lesson) called *password*. For now, just think of a variable as a name for a container to store some information in. Once the user's chosen password has been stored, the algorithm must request input from the user. If the input does not match the password the user originally set, the algorithm must store the incorrect passwords in a list until the correct password is entered. At that point, it must print out the value of the variable *password* (i.e., the user's chosen password that has been stored in the password variable/container), as well as the incorrect passwords:

Pseudocode solution:

```
request input from the user
store input into variable called 'password'
request second input from the user
if the second input is equal to 'password'
    output the 'password' and the incorrect inputs (which should be none
    at this point)
if the second input is not equal to 'password'
    repeatedly request input until input matches 'password'
    store the non-matching input for later output
when the input matches 'password'
    output 'password'
    and output all incorrect inputs.
```

In the Practical Tasks at the end of this lesson, you will practise creating high-level solutions to simple problems using pseudocode, as seen above.



A note from the Hyperion Team



Did you know that although Python is named after the Monty Python comedy group, it was created by '[Benevolent Dictator For Life](#)' Guido van Rossum in 1991, who now works for Microsoft? At the time when he began implementing the Python language, he was also reading the published scripts from *Monty Python's Flying Circus* (a BBC comedy series from the seventies). His inspiration for the name came from the comedy series, and his motivation for creating the Python language stemmed

from the desire to create a simple scripting language drawing on his experience with the ABC programming language.

INTRODUCTION TO PYTHON

Python is a powerful, widely used programming language. In comparison with Java, Python is a more recent, efficient, and arguably faster programming language. The syntax (the way the code is written) is, nonetheless, very similar to Java.

Here are a few more reasons to use Python:

- **Simple, yet powerful:** Looking at languages like C++ and Java can flummox and scare the beginner, but Python is intuitive, with a user-friendly way of presenting code. Python's succinctness and economy of language allow for speedy development and less hassle over useful tasks. This makes Python easy on the eyes and mind.
- **From child's play to big business:** While Python is simple enough to be learned quickly (even by kids), it is also powerful enough to drive many big businesses. Python is used by some of the biggest tech firms, such as

Google, Yahoo!, Instagram, Spotify, and Dropbox, which should speak volumes about the job opportunities out there for Python developers.

- **Python is on the web:** Python is a very appealing language of choice for web development. Sites such as Pinterest and Instagram make use of the versatility, rapidity, and simplicity of Django (a web development framework written in Python).
- **Even Dropbox was built using Python:** Dropbox must save massive quantities of files while supporting a similarly massive degree of user growth. Did you know that 99.9% of Dropbox code is written in Python? Using Python has helped Dropbox gain more than a hundred million users. With only a few hundred lines of Python code, they were able to scale their user numbers dramatically, proving the utility of the language!



Know this – Python is hot!

The demand for Python programmers is only growing. Python boasts the highest year-on-year increase in terms of demand by employers (as reflected in job descriptions online) as well as popularity among developers. Python developers are one of the highest-paid categories of programmers! The demand for Python is only set to grow further with its extensive use in analytics, data science, and machine learning.



THE ZEN OF PYTHON

The *Zen of Python*, written in 1999 by Tim Peters, mentions all the software principles that influence the design of the Python language.

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one – and preferably only one – obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

*Although never is often better than **right** now.*

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea – let's do more of those!

Ever need to recall these principles? Try entering this into your Python interpreter:

import this

WHAT IS PROGRAMMING?

Programmers write code statements to create *programs*. Programs are executable files that perform the instructions given by the programmer.

Code can be written in different programming *languages*, such as Python, Java, JavaScript, and C++. In this course, you will start by learning Python.

After writing Python code, you need to save it in a Python file. A Python file has the following file naming format:

filename.py

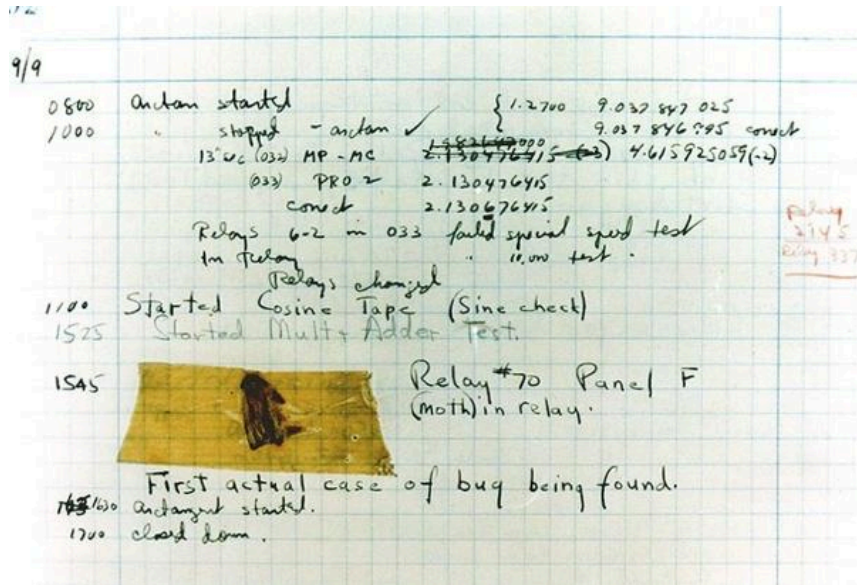
The filename can be any valid filename and **.py** is the file extension.

You can then 'run' the Python file. In this process, the Python program you have written is executed and displays the outcomes that may result based on what the code statements say. Information about how to run Python files is given in the example file (**example.py**) that accompanies this task. We will now show you how to write some basic code in Python and perform some basic operations.



Sorry to interrupt, but did you know that the first computer 'bug' was named after a real bug? Yes, you read that right! While using 'bug' to mean a technical error was first coined by Thomas Edison in 1878, it was only 60 years later that someone else popularised the term.

In 1947, Grace Hopper, a US Navy admiral, recorded the first computer 'bug' in her logbook. She was working on a Mark II computer when a moth was discovered stuck in a relay, hindering the operation. She proceeded to remove the moth, thereby 'debugging' the system, and taped it into her logbook! In her notes, she wrote, 'First actual case of a bug being found.'



Riaz Moola, Founder and CEO

THE PRINT() FUNCTION

You may want your program to display or output information to the user. The most common way to view program output is to use the **print** function. To use **print**, we enter the **print** command followed by one or more arguments in brackets. Let's first take a moment to understand what arguments are, as well as two other new terms: parameters and variables.

Remember from earlier in this task that a **variable** is a computer programming term that is used to refer to the storage locations for data in a program. Each variable has a name that can be used to refer to some stored information known as a **value**.

Functions and methods in Python are blocks of code that perform one or more specific functions. A **parameter** is a variable in a function definition that tells you the sort of data you will need to give the function to work with when it runs. When a function is actually called (in other words, the programmer instructs the block of code making up the function to run), the **arguments** are the *data you pass into* the function's parameters (i.e., the actual data you give to the function to work on when it runs). The same is true of a method.

Now that you understand these basic concepts, let's review how **print** works. We enter the **print** command into a Python file followed by one or more arguments in

brackets. In programming, a **command** is an instruction given by a user telling a computer to do something. Together, a command and an argument are known as a **statement**. Consider the Python statement below:

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

When you run this program, the computer will output the argument “**Hello, World!**” that was passed into the input parameter. Note that the argument is enclosed in double quotes (“”). This is because “Hello, World!” is a type of variable called a **string**, i.e., a list of characters. You’ll learn more about strings and other variable types later in this task.

Note that the Python shell (the window that is displayed when you run a Python program) only shows the output of the program. Other statements in your code that don’t create output will be executed but not displayed in the Python shell.

SYNTAX RULES

All programming languages have *syntax* rules. Syntax is the ‘spelling and grammar setup’ of a programming language and determines how you write correct, well-formed code statements. If you make a mistake by breaking the ‘spelling and grammar’ rules for code in Python, this is called a syntax error.

A common syntax error you could make in the **print** statement we looked at above is forgetting to add a closing quotation mark at the end of the string parameter ("). In strings, all opening quotation marks require a closing one – the opening one shows the string is starting, and the closing one shows where it ends. Another common syntax error that you could make in the **print** example above is forgetting to add a closing bracket ')'. Remember that all opening brackets '(' require a matching closing one, ')'!

Any program you write must be exactly correct. All code is case-sensitive. This means that ‘Print’ is not the same as ‘print’. If you enter an invalid Python command, misspell a command, or misplace a punctuation mark, you will get a syntax error when trying to run your Python program.

Errors appear in the Python shell when you try to run a program and it fails. Be sure to *read all errors carefully* to discover what the problem is. Error reports in the Python shell will even tell you what line of your program had an error. The process of resolving errors in code is known as *debugging*.

HOW TO GET INPUT

Sometimes you want a user to enter data through the keyboard that will be used by your program. To do this, you use the **input** command.

When the program runs, the **input** command, which can be seen in the example below, will show the text "Enter your name: " in the output box of the program. The program will then halt until the user enters something with their keyboard and presses enter.

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

The variable *name* stores what the user entered into the box as a **string**. Storing and declaring variables doesn't produce any output.

WHAT ARE VARIABLES?

Let's consider variables in a little more depth. To be able to perform calculations and instructions, we need a place to store values in the computer's memory. This is where variables come in. A *variable* is a way to store information. It can be thought of as a type of 'container' that holds information. In the example above, the input command was used to tell the computer to take whatever the user typed before they pressed enter and place that in a container as a variable of the **string** type.

Variables in programming work the same as variables in mathematics. We use them in calculations to hold values that can be changed. In maths, variables are named using letters, like *x* and *y*. In programming, you can name variables whatever you like, as long as you don't pick something that is a keyword (also known as a 'reserved' word) in the programming language. It is best to name them something useful and meaningful to the program or calculation you are working on. For example, **num_learners** could contain the number of learners in a class, or **total_amount** could store the total value of a calculation.

In Python, we use the following format to create a variable and assign a value to it:

```
variable_name = value_you_want_to_store
```

Check out this example:

```
num = 2
```

In the code above, the variable named **num** is assigned the integer (or whole number) 2. Hereafter when you type the 'word' **num**, the program will refer to the appropriate space in memory where the variable is stored, and retrieve the value 2 that is stored there.

We use variables to hold values that can be changed (can vary). You can name a variable anything you like as long as you follow the rules shown below. However, as previously stated, giving your variables meaningful names is good practice.

Below are examples of bad naming conventions vs good naming conventions:

- **my_name** = "Tom" # Good variable name
- **variableOne** = "Tom" # Bad variable name
- **string_name** = "Tom" # Good variable name
- **h4x0r** = "Tom" # Bad variable name

Here, **my_name** and **string_name** are examples of descriptive variables as they reveal what they are and what content they store, whereas **variableOne** and **h4x0r** are terrible names because they are not descriptive.



A note from the
Hyperion Team

Now that you are a little more familiar with Python and creating basic programs, we would like to show you some stuff to help you on your journey to becoming a seasoned programmer.

Creating excellent content requires good tools and equipment. This applies equally well to programming. There are some great tools and resources available online that you can start using as soon as possible if you have not already, to make the coding process just that much more convenient. Head to the [HyperionDev Blog](#) where you will find essential utilities and resources for programmers.

NAMING VARIABLES

Variable naming rules

As previously mentioned, it is very important to give variables descriptive names that reference the value being stored. Here are the naming rules in Python (these can differ in other programming languages):

1. Variable names must start with a letter or an underscore.
2. The remainder of the variable name can consist of letters, numbers, and underscores.
3. Variable names are case sensitive so 'Number' and 'number' are each different variable names.
4. You cannot use a Python keyword (reserved word) as a variable name. A reserved word has a fixed meaning and cannot be redefined by the programmer. For example, you would not be allowed to name a variable *print* since Python already recognises this as a keyword. The same is true of the keyword *input*.

Variable naming style guide

The way you write variable names will vary depending on the programming language you are using. For example, the **Java** style guide recommends the use of camel case – where the first letter is lowercase, but each subsequent word is capitalised with no spaces in between (e.g., `thisIsAGoodExampleOfCamelCase`).

The style guide provided for **Python** code, [PEP 8](#), recommends the use of snake case – all lowercase with underscores in between instead of spaces (e.g., `this_is_a_good_example_of_snake_case`). You should use this type of variable naming for your Python tasks.

In maths, variables only deal with numbers, but in programming, we have many different types of variables and many different types of data. Each variable data type is specifically created to deal with a specific type of information.

VARIABLE DATA TYPES

There are five major types of data that variables can store. These are **strings**, **chars**, **integers**, **floats**, and **booleans**.

- **string:** A string consists of a combination of characters. For example, it can be used to store the surname, name, or address of a person. It can also store numbers, but when numbers are stored in a string you cannot use them for calculations without changing their data type to one of the types intended for numbers.
- **char:** Short for **char**acter. A char is a single letter, number, punctuation mark, or any other special character. This variable type can be used for storing data like the grade symbol (A–F) of a pupil. Moreover, strings can be thought of (and treated by functions) as lists of chars in situations in which this approach is useful.
- **integer:** An integer is a whole number or number without a decimal or fractional part. This variable type can be used to store data like the number of items you would like to purchase, or the number of students in a class.
- **float:** We make use of the float data type when working with numbers that contain decimals or fractional parts. This variable type can be used to store data like measurements or monetary amounts.
- **boolean:** Can only store one of two values, namely TRUE or FALSE.

The situation you are faced with will determine which variable type you need to use. For example, when dealing with money or mathematical calculations you would likely use **integers** or **floats**. When dealing with sentences or displaying instructions to the user you would make use of **strings**. You could also use **strings** to store data like telephone numbers that are numerical but will not be used for calculations. When dealing with decisions that have only two possible outcomes, you would use **booleans**, as the scenario could only either be true or false.

Variables store data and the type of data that is stored by a variable is intuitively called the *data type*. In Python, we do not have to declare the data type of the variable when we declare the variable (unlike certain other languages). This is known as ‘weak typing’ and makes working with variables easier for beginners. Python detects the variable's data type by reading how data is assigned to the variable as follows:

- Strings are detected by quotation marks " ".
- Integers are detected by the lack of quotation marks and the presence of digits or other whole numbers.
- Floats are detected by the presence of decimal point numbers.
- Booleans are detected by being assigned a value of either 'True' or 'False'.

So, if you enter numbers, Python will automatically know you are using integers or floats. If you enter a sentence, Python will detect that it is storing a string. If you want to store something like a telephone number as a string, you can indicate this to Python by putting it in quotation marks, e.g. `phone_num = "082 123 4567"`.

You need to take care when setting a string with numerical information. For example, consider this:

```
number_str = "10"
print(number_str*2) # Prints 1010- prints string twice
print(int(number_str)*2) # Prints 20 because the string 10 is cast to number 10
```

Watch out here! Since you defined 10 within quotation marks, Python figures this is a string. It's not stored as an integer even though 10 is a number, as numbers can also be made into a string if you put them between quotation marks. Now, because 10 is declared as a string here, we will be unable to do any arithmetic calculations with it – the program treats it as if the numbers are letters. In the above example, when we ask Python to print the string times 2, it helpfully prints the string twice. If we want to print the value of the number 10 times 2, we have to cast the string variable to an integer (convert it) by writing `int(number_string)`. Take heed that all variable types can be converted from one to another, not just ints and strings.

There is also a way that you can determine what data type a variable is, using the `type()` built-in function. For example:

```
mystery_1 = "10"
mystery_2 = 10.6
mystery_3 = "ten"
mystery_4 = True

print(type(mystery_1))
print(type(mystery_2))
print(type(mystery_3))
print(type(mystery_4))
```

Output:

```
<class 'str'>
<class 'int'>
<class 'str'>
<class 'bool'>
```

The output shows us the data type of each variable in the inverted commas.

CASTING

Let's return to the concept of changing variable types from one to another. In the string-printing example above, you saw something we called *casting*. Casting means taking a variable of one particular data type and 'turning it into' another data type. Putting the 10 in quotation marks will automatically convert it into a string, but there is a more formal way to change between variable types. This is known as *casting* or type conversion.

Casting in Python is pretty simple to do. All you need to know is which data type you want to convert to, and then you can use the corresponding function.

- **str()** – converts a variable to a string
- **int()** – converts a variable to an integer
- **float()** – converts a variable to a float

```
number = 30
number_str = "10"
print(number + int(number_str)) #Prints 40
```

This example converts **number_str** into an integer so that we can add two integers together and print the total. We cannot add a string and an integer together. (Are you curious what would happen if you didn't cast **number_str** to a string? Try copying the code in the block above, pasting it into VS Code, and running it. What happens?)

You can also convert the variable type entered via **input()**. **By default, anything entered into an input() is a string.** To convert the input to a different data type, simply use the desired casting function.

```
num_days = int(input("How many days did you work this month?"))
pay_per_day = float(input("How much is your pay per day?"))
salary = num_days * pay_per_day
print("My salary for the month is USD:{}".format(salary)) //explanation below
```

When writing programs, you'll have to decide what variables you will need.

Take note of what is in the brackets on line four above. When working with strings, we are able to put variables into our strings with the *format* method. To do this, we use curly braces `{}` as placeholders for our values. Then, after the string, we put `.format(variable_name)`. When the code runs, the curly braces will be replaced by the value in the variable specified in the brackets after the format method.

Working with the f-string

The f-string is another approach to including variables in strings. The syntax for working with the f-string is quite similar to what is shown above in the format method example.

Notice that we declare the variables upfront, and we don't need to tag on the `.format` method at the end of our string the way we did when using the format approach. Also, note the **f** at the beginning of the string:

```
num_days = 28
pay_per_day = 50
print(f"I worked {num_days} days this month. I earned ${pay_per_day} per day.")
```

Output:

```
'I worked 28 days this month. I earned $50 per day.'
```

f-strings provide a less verbose way of interpolating (inserting) values inside string literals. You can read more about in the [Python documentation](#).

If you wanted to use the `str.format()` method in the same scenario as the f-string example, you could do so as follows:

Example 1: insert values using index references

```
print("You worked {0} this month and earned ${1} per day".format(num_days = 22, pay_per_day = 50))
```

Example 2: insert values using empty placeholders

```
print("You worked {} this month and earned ${} per day".format(num_days = 22, pay_per_day = 50))
```

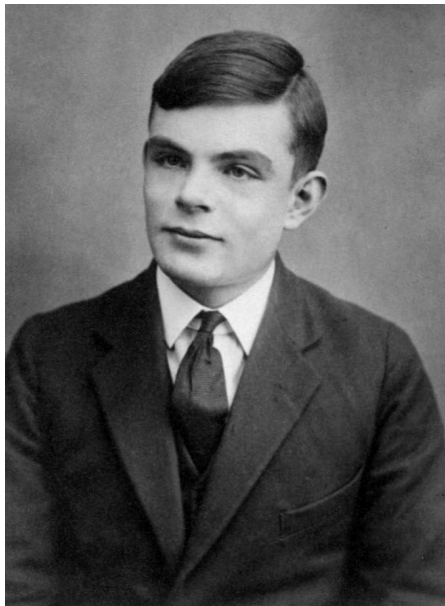
What do you think the advantage might be of using index references?



A note from the Hyperion Team

Father of modern-day computing

Have you heard about Alan Turing?



Alan Turing (1912–1954) was a British mathematician, logician, and cryptographer. He is considered by many to be the father of modern computer science. He designed and built some of the earliest electronic, programmable, digital computers.

During the Second World War, Alan Turing was recruited by the military to head a classified mission at Bletchley Park. This mission was to crack the Nazi's Enigma machine code that was used to send secret military messages. Many historians believe that breaking the Enigma code was key to bringing the war to an end in Europe. Turing published a paper in 1936 that is now recognised as the foundation of computer science.

Source: [Wikipedia](#)

Now it's time to try your hand at a **Practical Task**.

Instructions

- The directory called 'Examples' contains examples of a batch file (if you are using Windows) and a shell script called 'MacExample.sh' (if you are using Linux or macOS). Please read through the comments in the example file that is **relevant to you** (based on the operating system you are running on your PC) before attempting this task.
- The additional reading (Yang & Tamuri, 2015) is optional reading. However, we do recommend that you consult this brief guide for further information if you get stuck.

Practical Task 1

Follow these steps:

- Create a script file called **file_cd**. (Remember to save this file with a .bat extension if you are using Windows.)
 - Inside **file_cd**, insert commands to create three new folders (directories). Name your folder as you wish.
 - Next, insert commands to navigate inside one of the folders you created and create three new folders inside this folder. Also, insert commands to remove two of the folders you created.
- Create a batch file called **ifExample** (again, remember to save this file with a .bat extension if you are using Windows). Inside it, add an if statement to make a new folder called **if_folder** if one of the folders you created is named **new_folder**.
- Next, add an if-else statement that makes a new folder called **hyperionDev** if an **if_folder** exists, or else make a new folder called **new-projects** if it does not.

You will need to employ the following syntax for these conditional statements:

If exist *filename* *command*

If exist *filename* (*command*) else (*command*)

- Please continue with the rest of this task and complete the remaining Practical Tasks.

This lesson is continued in the example files (**example_first_program.py** and **example_variables.py**) provided in this task folder. Open these files using VS Code. The context and explanations provided in the examples should help you better understand some simple basics of Python.

You may run the examples to see the output. The instructions on how to do this are inside each example file. Feel free to write and run your own example code before attempting the task to become more comfortable with Python.

Try to write comments in your code to explain what you are doing in your program (read the example files for more information, and to see examples of how to write comments).

You are not required to read the entirety of **Additional Reading.pdf**. It is purely for extra reference. That said, do take a look – you could find it very useful!

Practical Task 2

Follow these steps:

- Create a new Python file in the Dropbox folder for this task, and call it **hello_world.py**.
- First, provide pseudocode as comments in your Python file, outlining how you will solve this problem (you'll need to read the rest of this Practical Task first of course!).
- Now, inside your **hello_world.py** file, write Python code to take in a user's name using **input()** and then print out the name.
- Use the same input and output approach to take in a user's age and print it out.
- Finally, print the string "Hello World!" on a new line (the new line will happen by default if you use a separate print statement to the one you used immediately above to print out the age, because each print statement automatically inserts an 'enter', or newline instruction, at the end).

Practical Task 3

Follow these steps:

- Create a new Python file in the Dropbox folder for this task, and call it **details.py**.
- As in Practical Task 2, please first provide pseudocode as comments in your Python file, outlining how you will solve this problem.
- Use an **input()** command to get the following information from the user:
 - Name
 - Age
 - House number
 - Street name
- Print out a single sentence containing all the details of the user.
- For example:

```
This is John Smith. He is 28 years old and lives at house  
number 42 on Hamilton Street.
```

Practical Task 4

Follow these steps:

- Create a new Python file in this folder called **conversion.py**.
- As in the previous Practical Tasks, please first provide pseudocode as comments in your Python file, outlining how you will solve this problem.
- Declare the following variables:
 - **num1 = 99.23**
 - **num2 = 23**
 - **num3 = 150**
 - **string1 = "100"**
- Convert them as follows:
 - **num1** into an integer

- `num2` into a float
- `num3` into a string
- `string1` into an integer
- Print out all the variables on separate lines.



Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

[Click here](#) to share your thoughts anonymously.

REFERENCES

Yang, Z. & Tamuri, A. (2015). *Getting Started with Mac OS X/ Linux Command Terminal*.
<http://abacus.gene.ucl.ac.uk/software/CommandLine.MACosx.pdf>