Hyperiondev

TASK

Logical Thinking through Pseudocode and Algorithm Design

Visit our website

Introduction

WELCOME TO THE HYPERIONDEV DATA SCIENCE BOOTCAMP!

Data science is a rapidly evolving field and has applications in practically every industry. As ever-increasing volumes of data get generated, stored and used for informing strategic decisions, there is tremendous value in being able to make sense of raw data and gathering meaningful insights from it.

That is what makes data science ubiquitous. Once you understand how to think like a data scientist and work with data using popular tools and techniques, you will be able to apply your learnings in sectors as diverse as marketing, health, finance, technology, sports and public policy.

As a data scientist, you will often analyse large amounts of structured and unstructured data for purposes such as identifying patterns, predictive modelling, problem-solving and visual storytelling. In doing so, you will draw upon your knowledge of concepts and techniques from mathematics, statistics and computer science. If you have a curious bent of mind, enjoy problem-solving, and aren't afraid of numbers, this may be the career for you.

This bootcamp is divided into three levels of study:

1. **Python for Data Science (Beginner level)** – This first level helps you understand the fundamentals of Python, which is fast emerging as the most popular programming language for data science.

2. **Data Analytics and Exploration (Intermediate level)** – In this level, you will learn how to work with databases and popular Python packages to handle a broad set of data analysis problems. You also learn how to create visualisations that can communicate insights about your data.

3. **Machine Learning (Advanced level)** – In this final stretch of your bootcamp, you will begin with fundamental statistical and machine learning concepts. Progressing through a series of lessons will help you build a solid understanding of supervised learning, unsupervised learning and machine learning applications in various industries.

The bootcamp is structured as a series of 'tasks'. Tasks are designed to teach you the theory needed to develop your skills, as well as give you the platform to apply your newly-gained knowledge by completing practical exercises.

Get in touch
Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to log in to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

LOGICAL THINKING THROUGH PSEUDOCODE AND ALGORITHM DESIGN

In this first task, we will explore how to think logically by delving into pseudocode and algorithm design.

Logical thinking is 'the process in which one uses reasoning consistently to come to a conclusion'. The ability to think logically is crucial for data scientists as they need to ask the right questions, plan and execute a series of rational exercises, and eventually draw the right conclusions when presented with vast amounts of structured and unstructured data.

The ability to think logically is also necessary for programming, as programmers often have to visualise a problem and figure out how to implement the steps to solve it. This is where pseudocode comes in. Pseudocode is a simple way of writing programming code in plain English and does not obey any specific syntax rules. Pseudocode is easy to write and understand even if you have no coding experience, as you simply need to write down a logical breakdown of what you want your computer program to do.

Understanding how to think through a problem and how to write pseudocode or an algorithm to solve that problem will ease your transition into the world of programming and data science. Learning the fundamentals of Python – a very popular programming language also used widely by data scientists – is the first step in this bootcamp, as you will later rely on your knowledge of Python to use

various data science libraries in the intermediate and advanced levels of this bootcamp.

A note from the

HyperionDev Team

The key to becoming a competent programmer is utilising all available resources to their full potential. Occasionally, you may stumble upon a piece of code that you cannot wrap your head around. So, knowing which platforms to go to for help is important!

HyperionDev knows what you need, but there are also other avenues you can explore for help. One of the most frequently used by programmers around the world is **Stack Overflow**.

Also, **our blog** is a great place to find detailed articles and tutorials on concepts into which you may want to dig deeper. For instance, after being introduced to machine learning fundamentals later in this bootcamp, you may want to **read up further about machine learning on our blog** or learn how to **deploy a machine learning model with Flask**.

Our blog is updated frequently, so be sure to check back from time to time for new articles or subscribe to updates through our **Facebook page**.

PSEUDOCODE

Pseudocode is a step-by-step description in plain English of what a computer program or algorithm must do, which can eventually be converted into a programming language. Pseudocode is used by programmers in the planning of computer program development to initially sketch out their code before writing the actual code.

Pseudocode makes creating programs easier. Program development can be complex and long, so preparation is key. Unless you understand the complete flow of a program, it is difficult to anticipate and account for possible errors.

There is no specific convention for writing pseudocode, as **a program in pseudocode cannot be executed**. In other words, pseudocode itself is not a programming language; it just makes the eventual coding a little simpler. It is intended for human reading rather than machine reading and is easier to

understand than conventional programming language code. Pseudocode does not need to obey any specific syntax rules (unlike conventional programming languages) and hence can be understood by any programmer, irrespective of the programming languages they're familiar with.

Although pseudocode does not have defined syntax rules, it's best to keep the following norms in mind:

- Use a plain-text editor (like Notepad) to write the pseudocode. Remember that pseudocode cannot be executed, so you will not be 'running' it after writing it down.
- Use clear language that can be easily transcribed into instructions for the computer.
- Restrict yourself to writing only one statement per line.
- Use whitespace and indentation to improve the readability of your pseudocode:
    - Sections of the pseudocode that involve obtaining an input from the user (for example, asking the user to enter a number) should all be indented similarly, while the next section of the pseudocode which discusses further instructions or the output should be indented further to the right;
    - When writing an IF condition (which means that a particular instruction will be carried out if the given condition is true, and another instruction will be carried out if the given condition is false), indent the instructions further to the right compared to the IF condition.
- Make sure you are taking into account all possible scenarios and are describing everything that is happening in the process.

EXAMPLES OF PSEUDOCODE

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. Take a look at some examples of pseudocode below.

Example 1: An algorithm that prints out passed or failed, depending on whether a student's grade is greater than or equal to 50.

Pseudocode:

```
if grade is equal to or greater than 50
    print "passed"
```

```
else
      print "failed"
```

Example 2: An algorithm that asks a user to enter their name and prints out "Hello, World" if their name is "John".

Pseudocode:

```
request user's name
if the inputted name is equal to "John"
     print "Hello, World"
```

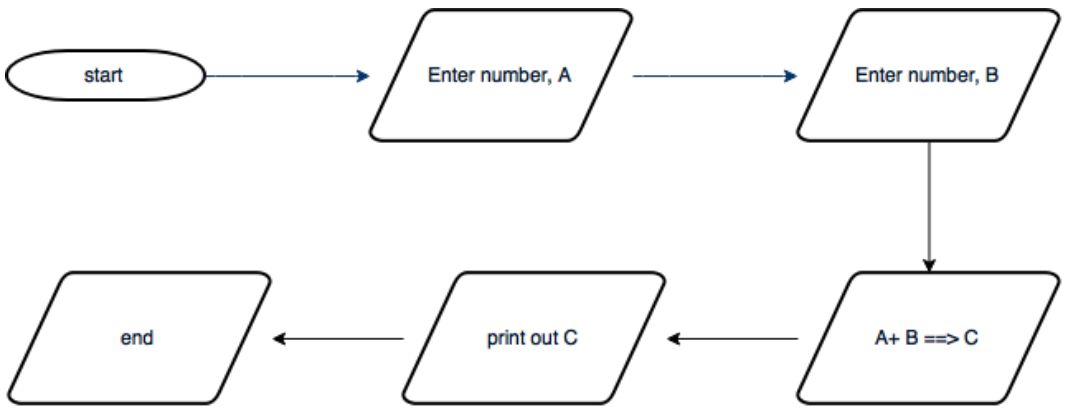
Example 3: An algorithm that requests an integer from the user and prints "fizz" if the number is even or "buzz" if it is odd.

Pseudocode:

```
request an integer from the user
if the integer is even
      print "fizz"
else if the integer is odd
      print "buzz"
```

Flowcharts can be thought of as a graphical implementation of pseudocode. This is because a flowchart is more visually appealing and probably more readable than a “text-based” version of pseudocode.

For example, for an algorithm that prompts a user to enter two integers consecutively, and then proceeds to sum up the two inputs and print out the result, the flowchart would look something like this:



ALGORITHM DESIGN AND REPRESENTATION

What exactly 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. This trick is a general solution for multiplying any single-digit number by 9. That's an algorithm!

This process of designing algorithms is interesting, intellectually challenging, and a core part of programming. An algorithm should follow a certain set of criteria so that it can be easily readable not only to yourself but to third parties reading your work. Clear and concise writing of algorithms is reflective of an organised mind.

An algorithm should usually have some input and, of course, some eventual output. Input and output help the user keep track of the current status of the program. It also aids in debugging if any errors arise. Say you have a series of calculations in your program that build off each other. It would be helpful to print out each of the programs to see if you're getting the desired result at each stage. Therefore, if a particular sequence in the calculation is incorrect, you would know exactly where to look and what to adjust.

Ambiguity is another criterion to take into consideration in the development of algorithms. Ambiguity is a type of uncertainty of meaning in which several interpretations are plausible. Your algorithms need to be as clear and concise as possible to prevent unintended outcomes.

Last but not least, you should keep in mind that computing resources (such as your machine's Random Access Memory or RAM) are finite. Some programs may require more RAM than your computer has, or take too long to execute. Therefore, codes must be written efficiently so that the load on your machine is minimised.

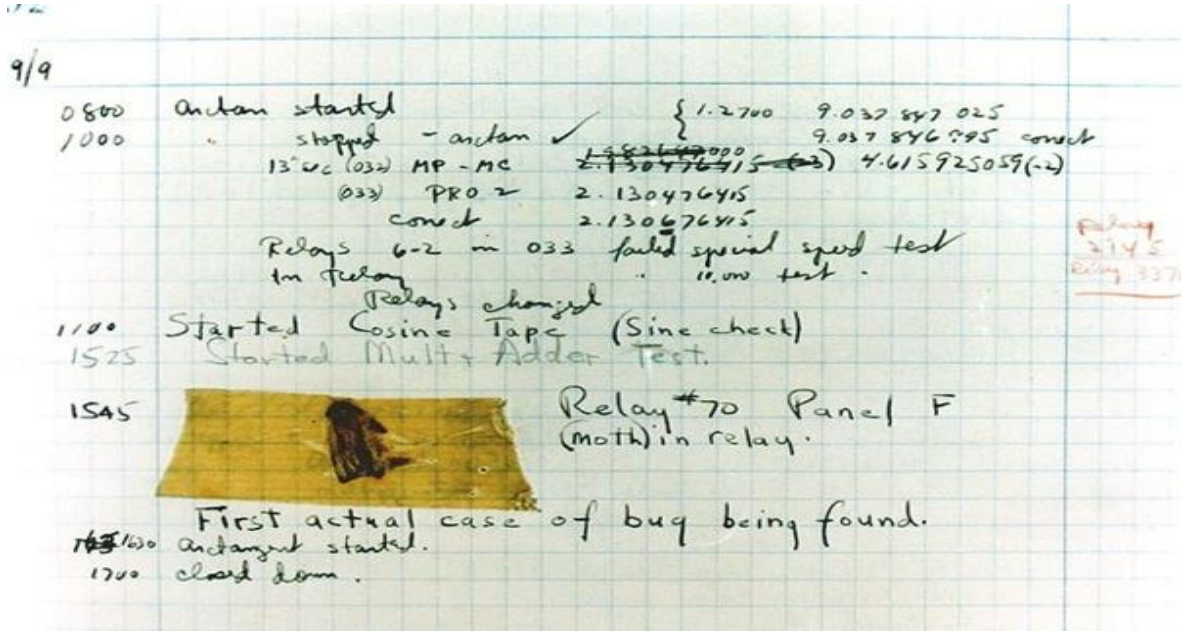
A note from our coding mentor
Sarah

Did you know that the first computer "bug" was named after a real bug? Yes, you read that right!

While the term "bug" in the meaning of 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 as she was working on a Mark II computer. A moth was discovered stuck in a relay and thus hindering the operation. She proceeded to remove the moth, thereby 'debugging' the system, and taped it in her logbook.

In her notes, she wrote, "First actual case of bug being found."



(**Source**)

THINKING LIKE A PROGRAMMER

Thus far, you've covered concepts that will see you starting to think like a programmer. But what exactly does thinking like a programmer entail?

The single most important skill for a computer scientist is problem-solving. Problem-solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly and accurately. The process of learning to program is an excellent opportunity to practice problem-solving skills.

One of the greatest misconceptions about programming — certainly at introductory levels — is that programming is riddled with mathematics. If you have the idea that programming will require going back to the school days of battling with trigonometry, algebra and the like, you're wrong. There is very little mathematics involved in programming. So, if you don't exactly love mathematics, don't be discouraged!

Programming merely involves the use of logic. The ability to think things through, understand the order in which they will take place, and have a sense of how to control that flow, pervades every aspect of programming. If you have an aptitude for logic, you're going to be in a good position to start wrestling with the task of programming.

Instructions

Read the examples provided above for writing pseudocode, and make sure you understand the process. You can also read these examples by opening the **example.py** file available within this task folder. Open it using Notepad++ (right-click the file and select 'Edit with Notepad++') or IDLE.

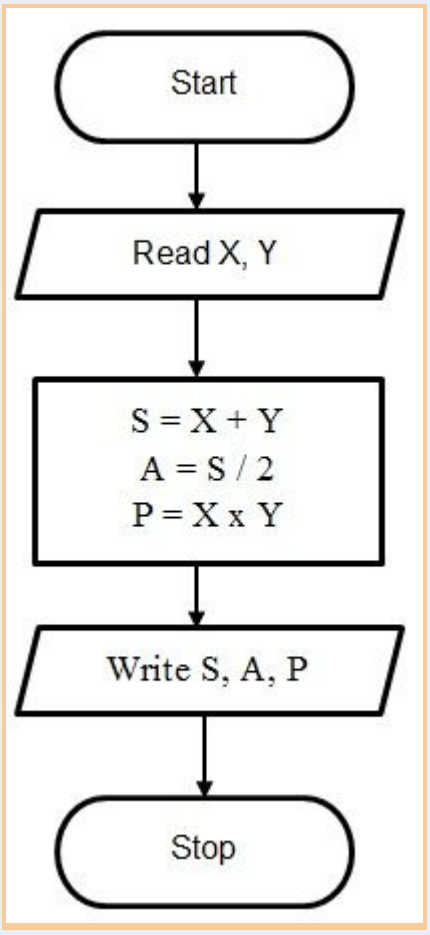
- **example.py** should help you understand some simple pseudo-code. Every task will have example code to help you get started. Make sure you read all of the examples and try your best to understand.
- This may take some time at first as it is slightly different from other languages, but persevere! Your mentor is here to assist you along the way.
- Feel free to write your own example code before doing the compulsory task to become more comfortable with some of the basic components.

Compulsory Task 1

Follow these steps:

- Create a new text file called **pseudo.txt** inside this task folder.
- Inside pseudo.txt, write pseudocode for each of the following scenarios:
    - An algorithm that asks a user to infinitely enter a positive number until the user enters a zero value, and then determines and outputs the largest of the inputted numbers.

    - An algorithm for the flowchart below:



Compulsory Task 2

Follow these steps:

- Create a new text file called **algorithms.txt** inside this folder.
- Inside **algorithms.txt,** write pseudocode for the following scenarios:
    - An algorithm that requests a user to input their name and then stores their name in a variable called first_name. Subsequently, the algorithm should print out first_name along with the phrase “Hello, World”.
    - An algorithm that asks a user to enter their age and then stores their age in a variable called age. Subsequently, the algorithm should print out “You’re old enough” if the user’s age is over or equal to 18, or print out “Almost there” if the age is equal to or over 16, but less than 18. Finally, the algorithm should print out “You’re just too young” if the user is younger than (and not equal to) 16.

Compulsory Task 3

Follow these steps:

- We take academic integrity very seriously. Read through the additional reading on plagiarism in this task folder. Then create a file called **plagiarism.txt** and answer the following questions:

1. What is plagiarism? Explain it in your own words.
2. What are the four different types of plagiarism? Give an example of your own for each (i.e. not the same examples as the additional reading).
3. How can you avoid plagiarism in your work?

- Be sure to reference any sources you use in your answers!

Completed the task(s)?

Ask your mentor to review your work!

Review work

Thing(s) to look out for:

Make sure that you have installed and set up **Dropbox** correctly on your machine.

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.