HyperionDev

Pseudocode

Task

Visit our website

Introduction

This task will introduce 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. This is the reason why you will begin your adventure into programming with pseudocode.

Extra resources

If you're taking this course, you're brand new to the Python programming language. You may even be new to programming altogether. Either way, you've come to the right place!

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 essential.

HyperionDev knows what you need, but there are also other avenues you can explore for help. One of the most frequently used by programmers worldwide is **Stack Overflow**; it would be a good idea to bookmark this for future use.

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, when you get more comfortable with programming, if you want to **read up about machine learning**, or learn **how to deploy a machine learning model with Flask**, you can find all this information and more on our blog.

The 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

What is pseudocode, and why do you need to know about it? 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.

Algorithms

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. Imagine you're making a sandwich. There are some defined steps in the process:

1. Get two slices of bread.
2. Spread butter on one side of each slice.
3. Choose your filling. This could be cheese, ham, or anything you like.
4. Place the filling between the two slices of bread, butter side in.
5. Cut the sandwich in half if you like.
6. Eat and enjoy!

This is an algorithm because it's a clear set of steps to make a sandwich. You can use these steps for any type of sandwich, changing the filling as needed.

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.

Designing algorithms

The process of designing algorithms is interesting, intellectually challenging, and a core part of programming. But some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain how we do it, at least not in the form of an algorithm.

In some instances, you may be required to draft algorithms for a third party. Therefore, it is essential that your algorithms satisfy a particular set of criteria so that they can easily be read by anyone.

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

Input and output

Input can be data or information sent to the computer using an input device, such as a keyboard, mouse, or touchpad. Output is information that is transferred out from the computer, such as anything you might view on the computer monitor. It is sent out of the computer using an output device, such as a monitor, printer, or speaker.

Input and output help the user keep track of the current status of the program and they also aid in debugging if any errors arise. Debugging is the process of identifying and resolving errors, or "bugs", in software.

For example, you may have a series of calculations in your program that build on each other, so 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.

Take a look at the pseudocode example below, which deals with multiple inputs and outputs.

Problem

Write an algorithm that asks a user to input a password and then stores the password in a variable called "password". A variable is the simplest structure we use in coding to store values (you'll learn more about this soon). Subsequently, the algorithm requests input from the user. If the input does not match the password, it stores the incorrect passwords in a list until the correct password is entered, and then prints out the content of the variable "password" (i.e., the right password) and 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"

            print out the "password" and the incorrect inputs (which should be
none at this point)

if the second input is not equal to "password"

        request input until second input matches password

        store the non-matching input for later output

when second input matches "password"

        print out "password"

        and print out all incorrect inputs.
```

Variables

In a program, variables act as a kind of "storage location" for data. They are a way of naming or labelling information so that we can refer to that particular piece of information later on in the algorithm. For example, say you want to store the age of the user so that the algorithm can use it later. You can store the user's age in a variable called "age". Now, every time you need the user's age you can **input** the variable "age" to reference it.

As you can see, variables are very useful when you need to use and keep track of multiple pieces of information in your algorithm. This is just a quick introduction to variables, and you will get a more in-depth explanation later on in this course.

Writing algorithms

Now that you are familiar with variables, take a look at some of the important factors to keep in mind when writing algorithms for your pseudocode.

HyperionDev

Clarity

Clarity in algorithms is crucial. Your algorithm should be unambiguous, which means the algorithm should be straightforward and not open to multiple interpretations. Ambiguity occurs when a statement can be understood in various ways. You should make sure that your algorithms are clear and concise to avoid any misunderstandings or unexpected results.

Correctness

Algorithms must accurately address and resolve a specific set of problems, which is a concept known as correctness and generality. Your algorithm should execute without errors and effectively solve the problem it's designed to address.

Capacity

Last but not least, consider the capacity of your computing resources. Resources such as CPU power and memory are limited. Some tasks might require more memory than your computer has or could take too long to complete. It's important to find ways to write efficient code to reduce the load on your machine.

Pseudocode syntax

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 is a model of all programming languages that one uses to make eventual coding a little simpler.

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 to discuss, analyse, and agree on a program's design with a team of programmers, users, and clients before coding the solution.

In our provided examples, the pseudocode is indented, which improves readability and structure, especially when representing loops and conditional statements. Indentation, loops, and conditional statements are concepts that will be covered in the upcoming tasks. Although pseudocode isn't bound by strict rules, the use of indentation is a common formatting practice that mirrors many programming languages. You are encouraged to adopt this style early on to write clearer and more organised pseudocode.

Examples

Consider the following examples:

Example 1

An algorithm that prints out "passed" or "failed" depending on whether a student's grade is greater than or equal to 50 could be written in pseudocode as follows:

```
get grade

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" would be written as follows:

```
request user's name

if the input 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 would be written like this:

```
request an integer from the user

if the integer is even

    print "fizz"

else if the integer is odd

    print "buzz"
```

Example 4

An algorithm that calculates the grade average by inputting and adding up the grades of each of the 10 students and then prints out the class average, would be written as follows:

```
set the total class grade to zero

set counter to one

while counter is less than or equal to ten

        input the student's grade

        add the student's grade to the total class grade

calculate the class average by dividing the total grade sum by ten

print the class average
```
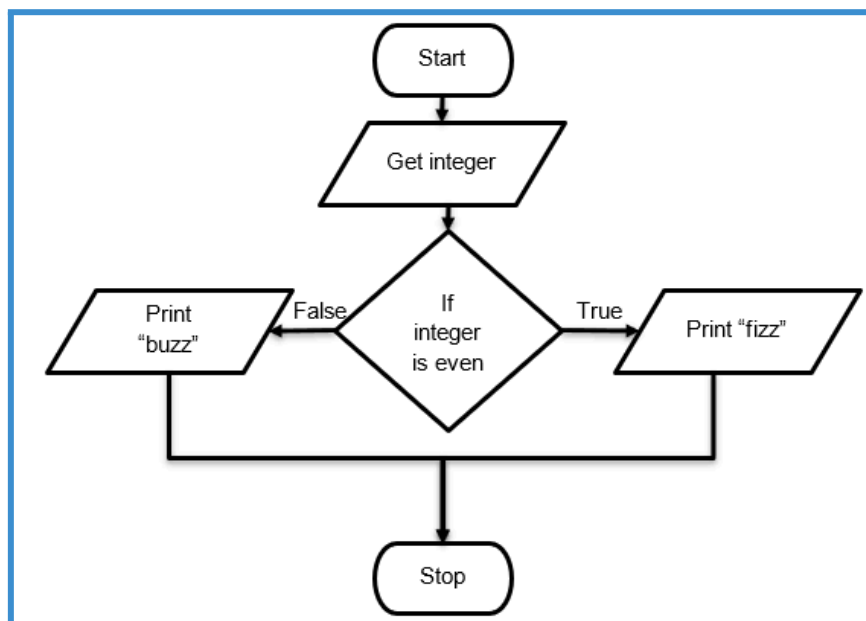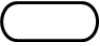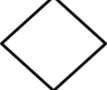
Flowcharts

Flowcharts can be thought of as a graphical implementation of pseudocode. They can be used because they are more visually appealing and probably more readable than a "text-based" version of pseudocode. An example of a flowchart that would visualise the algorithm for Example 3 above is shown in the following image:

Example 3 flowchart

There are a number of shapes or symbols used with flowcharts to denote the type of instruction or function you are using for each step of the algorithm.

The most common symbols used for flowcharts are shown in the table below:

Symbol	Use
⬭	Indicates the start and end of an algorithm.
▱	Used to get or display any data involved in the algorithm.
▭	Indicates any processing or calculations that need to be done.
◇	Indicates any decisions in an algorithm.

Think like a programmer

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

Well, this way of thinking combines some of the best features of mathematics, engineering, and natural science. Like mathematicians, computer scientists use formal languages to denote ideas (specifically computations). Like engineers, they design things, assemble components into systems, and evaluate tradeoffs among alternatives. Like scientists, they observe the behaviour of complex systems, form hypotheses, and test predictions.

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 solutions clearly and accurately. As it turns out, the process of learning to program is an excellent opportunity to practise problem-solving skills.

On one level, you will be learning to program, a useful skill by itself. On another level, you will use programming as a means to an end. As we go along, that end will become clearer.

Another important part of thinking like a programmer is actually thinking like a computer. The computer will run code in a logical step-by-step, line-by-line process. That means that it can't jump around. For example, if you want to write a program that adds two numbers together, you have to give the computer the numbers first before you instruct it to add them together. This is an important concept to keep in mind as you start your coding journey.
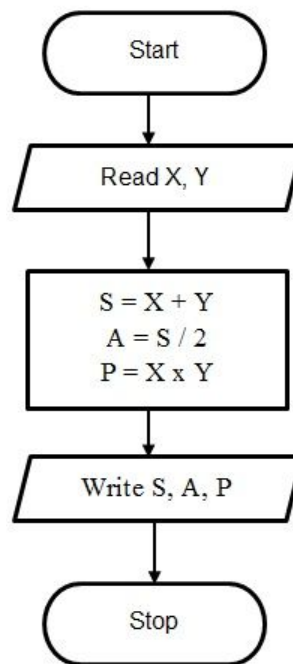
Code hack

Pseudocode is one of the most underrated tools a programmer has. It's worth getting into the habit of writing your thought process in pseudocode in a book or a separate file before you actually begin coding. This will help you make sure your logic is sound and your program is more likely to work!

Practical task

Follow these steps:

1. Use a plain text editor like Notepad++. Create a new text file called pseudo.txt inside the folder for this task.

2. Inside **pseudo.txt**, write pseudocode for each of the following scenarios:

   - An algorithm that asks a user to enter a positive number repeatedly until the user enters a zero value, then determines and outputs the largest of the numbers that were entered.
   - 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 reads an arbitrary number of integers and then returns their arithmetic average.
   - An algorithm that reads a grocery list and prints out the products (in alphabetical order) that are still left to buy.
   - An algorithm for the flowchart on the following page:

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.