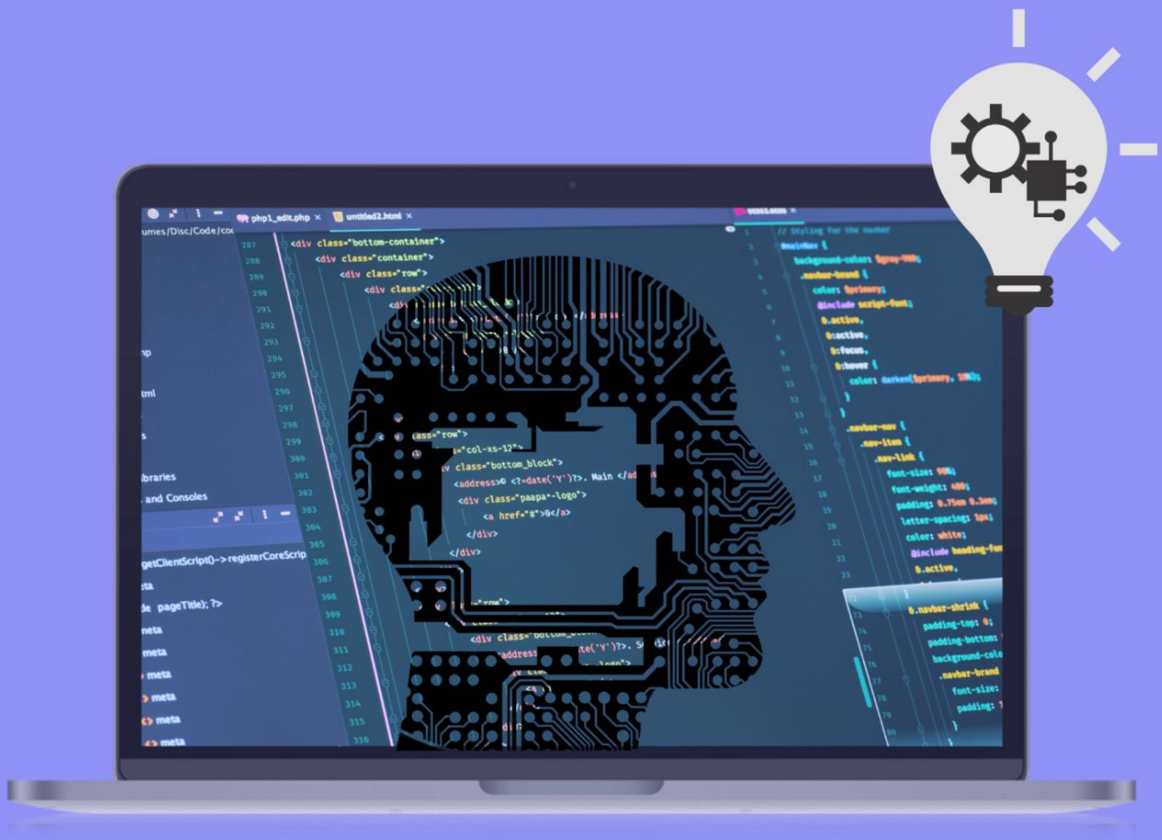
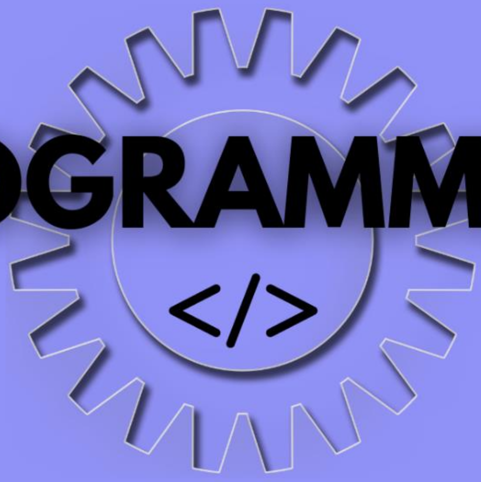


PROGRAMMING



ISAIAH A. CARRINGTON

Table of contents

Contents

Chapter 1: Introduction to the basics

This will be the first of several notes to explain the basics of programming and get you (the reader) started on your way to becoming a developer.

In this introduction to the basics, I will cover the following Concepts:

1. Introduction to Programming
2. Introduction to Problem Solving
3. Introduction to Algorithms
4. Narrative Writing
5. Introduction to Pseudocode
6. Introduction to Flowcharts

Introduction to Programming

Before we can begin learning how to program, we will have to learn what is programming.

What is programming?

Simply put, programming is the process of designing and building a computer program designed to accomplish a result or to perform a task.

How is this useful?

Programming has several applications in our modern day lives, solving several of our problems. For an easy example, think of Communication. By using programming, we have been able to create several applications which help to solve this problem, such as social media. Tasks that may have been long and tedious before by human hands, is now being automated by programming, and improving efficiency and accuracy by several times.

Note: I won't give a history of programming because that is not what this series of notes is about. This is simply a brief overview so you can understand what it is exactly that you are getting into too.

What part do programming languages play?

Programs in their most basic forms, is a sequence of instructions that a computer can follow. As you may know, computers do not understand languages like English. What they do understand however, are binary strings (i.e., a sequence of 0s and 1s). However, these binary strings are not easily understood by us humans. If I were to present to you a binary string and asked what it meant, by default you would have no clue. You may be able to figure it out by doing mathematical calculations, but especially for larger binary strings, it is by no means a simple feat. This is where programming languages come in.

The role of programming languages is to act as a sort of middle ground, between our understanding of languages (such as English), and the computer's understanding of binary strings (0s and 1s). Programming languages allow us to define instructions in a language (known as natural language. For example, English) that we understand, and then be able to automatically convert the instructions into machine code (0s and 1s) for the computer to understand and execute.

Deeper look at programming.

So now we know what a programming language is, let's take a deeper look.

There are a wide variety of programming languages that are available for us to use, each with their own strengths and weaknesses. With the great diversity among these programming languages, there exists a common ground for making programs. In this series, we will be looking at 3 of these methods, namely:

- Narratives
- Flowcharts
- Pseudocode

What's next?

Now that we have a basic understanding about what exactly programming is and why it is useful, now we can start learning more about the basics and getting started with applying it in real life.

To begin, we will start looking at problem solving.

Introduction to Problem Solving

For this section, we will look at the most fundamental part of programming, that is, Problem Solving.

What is Problem Solving?

Simply put, Problem Solving is solving a problem. In practice, it is a bit more complex than that.

Problem Solving involves taking a problem, understanding the problem and what it needs, and then making a solution that solves the problem completely.

How does it affect Programming?

Looking back, we should remember that programming exists so we can solve problems by use of computers. Stated more explicitly, programming allows us to take our solutions that we would have gained from Problem Solving the problem and implement it as a computer program.

In other words, without Problem Solving, programming loses one of its purposes.

How to Problem Solve?

Problem solving is not as difficult as some people make it out to be. Although the process may get increasingly harder as the problems get more difficult, the same steps can be applied to arrive at a solution. These steps are as follows.

1. Read the problem several times
2. Highlight the keywords in the problem
3. Break the problem down into simple steps
4. Find solutions for each step
5. Put it all together for your final solution

We'll be using this method of problem solving going forward for all examples that we do throughout this *document.*

Introduction to Algorithms

With the basic introduction to programming completed, we can begin looking at algorithms.

What is an Algorithm?

An algorithm can be defined as a sequence of clearly defined, explicit, unambiguous (clear) steps, designed to solve a problem in a finite time period. Algorithms are generally written in natural language, and as implied, provides a solution to a problem, which can then be implemented by use of a programming language.

How are they useful?

Algorithms allow us to see and theoretically test a solution before implementing it with a programming language. This can save time during the development process of the program. Also, as they are often written in natural language (Example: English), it is not dependent on any specific language. This means that turning the algorithm into a program is possible regardless of what language is being used.

Examples of Algorithms:

Examples of Algorithms include:

- Recipe to bake a cake
- Instructions to find the largest number of a given list
- Sorting numbers from smallest to largest
- Finding the shortest path through a maze

Ways of representing Algorithms.

As mentioned previously, we will be looking at 3 ways of representing algorithms: narratives, pseudocode, and flowcharts.

Introduction to Narrative Writing

As done previously, we will first start by identifying what a narrative is.

What is a narrative?

A narrative is the simplest way of representing an algorithm in natural language, void of any programming constructs. Due to its simplicity, narratives are also the best to show people who may not be programming savvy, when explaining how an algorithm may work. It is simply a sequence of natural language statements

How to write a narrative?

Writing a narrative is extremely simple. First, you take your problem, and understand what it is asking of you. After you understand what the problem is asking, then you write down each step to solving it in plain English. This will be understood better with the use of examples

The examples

Example 1:

Problem: “Write a narrative that will provide instructions on how to count only the even numbers between 1 and 15 inclusive”

Solution:

First, we read the problem and pick out keywords. For this problem, the keywords will be:

- **Even**
- **1**
- **15**

These are the words that instruct to us how the program is supposed to work. Now we turn it into a narrative by giving simple statements.

```
1.  Start counting upwards from 1
2.  Each time you increment, check to see if the number is an even number
3.  If the number is an even number, then display it to the screen
4.  Repeat steps 2 – 3 until the number is 15
```

Introduction to Pseudocode

So, we've looked at beginner information for Algorithms and Narratives. Understanding these basics will allow us to move onto the next stage… Pseudocode

What is Pseudocode?

The word Pseudo means "fake". So simply put, Pseudocode is fake code.

Pseudocode makes use of basic programming constructs and its' own syntax, which allows solutions to be drafted into a code like format. However, it is NOT a programming language, and therefore cannot be executed by a computer. Its main purpose is to provide a look at how a solution may look in a code like format, which makes translating to an actual programming language easier.

Note: Syntax can be thought of as the grammatical rules for a language. For Example, we have syntax for English, like using "a" before a word beginning with a consonant, and "an" before a word beginning with a vowel.

Using Pseudocode

As previously mentioned, pseudocode has its own syntax, that is, it has its own rules for how it should be written. We will look at some of the most common ones that we will be using throughout this book. Feel free to come back here at any time for a review.

Begin and End.

First off, are the BEGIN and END keywords. Like they sound, these keywords indicate the beginning and end of the Pseudocode. All complete pseudocode MUST contain these 2 keywords. In future examples, when I do not include these, that means that the example is a *snippet*, that is, only a section of the complete program.

Example:

```
BEGIN
  // Some Pseudocode statements here
END
```

Comments

Comments are text that we can use to describe what a piece of code, without having to worry about our program trying to run it. They are strictly for the readers of the code to get an understanding as to how a section of code works, or for some other explanation.

To use a comment in Pseudocode, we will make use of the **//** sign, as shown in the above *Begin and End* example.

Declare variable_name as type data_type

This is syntax used to declare variables (Remember this word. It will be explained in Chapter 2) with a given name as a specific data type. Don't mind if this sounds confusing now, it will make sense soon.

Example:

```
BEGIN
  DECLARE number as type integer
END
```

DISPLAY "…"

This statement says that everything after the keyword should be displayed to the screen.

Example:

```
BEGIN
  DISPLAY "Hello there"
END
```

READ variable_name

This statement says to get input from the user's keyboard, and when the user presses enter, to then save the input in the given variable name

Example:

```
BEGIN
  DISPLAY "Enter a number"
  READ number
  DISPLAY "Your number was ", number
END
```

Variable Assignment

In order to assign values to variables, we make use of the assignment operator, the **=** sign. Example:

```
BEGIN
  // This stores the number 10, in a variable called x
  x = 10
END
```

FOR variable_name = start_value TO end value DO… ENDFOR

This is the format for a **FOR** loop. Again, everything will make sense as we go in further detail. For now, an example

```
BEGIN
  DECLARE number as type integer
  FOR number = 1 to 10 DO
            // Do something in here
  ENDFOR
END
```

IF (condition) THEN… ENDIF

This is the syntax used for an If statement. As with the others, this will be explained later on in more detail. Example:

```
BEGIN
  DECALRE number as type integer
  number = 5
  IF number == 5 THEN
          DISPLAY "The number is 5"
  ENDIF
END

```

Putting it all together

Here, I will put together everything I would have covered in this section, to create a Pseudocode representation of the problem we look at in the narrative section. That is, displaying all even numbers between 1 and 15.

Example 1:

Problem: “Write a narrative that will provide instructions on how to count only the even numbers between 1 and 15 inclusive”

Solution:

First, we read the problem and pick out keywords. For this problem, the keywords will be:

- **Even**
- **1**
- **15**

These are the words that instruct to us how the program is supposed to work. Now we turn it into a narrative by giving simple statements.

```
1.  Start counting upwards from 1
2.  Each time you increment, check to see if the number is an even number
3.  If the number is an even number, then display it to the screen
4.  Repeat steps 2 – 3 until the number is 15
```

Figure 1: Example 1 of Narrative writing

Solution:

```
BEGIN
  DECLARE number as type integer
  FOR number = 1 TO 15 DO
          IF number % 2 == 0 THEN
                  DISPLAY number
          ENDIF
  ENDFOR
END
```

For now, I just want you to get a basic understanding of some of the concepts that we will be looking deeper into as we go forward. Don’t feel overwhelmed as it will all make sense soon.

Introduction to Flowcharts

What is a flowchart?

A flowchart can be defined as a visual representation of an algorithm. Flowcharts make use of various shapes and symbols to visual represent structures that exist within an algorithm.

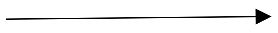
Why is a flowchart useful?

There are times when a visual representation might be more appropriate and easier to display and understand than a worded one.

Using a flowchart

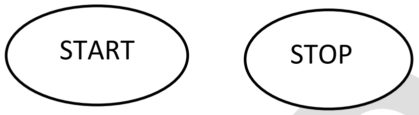
As mentioned, flowcharts use various symbols to convey different meanings. Here in this introduction, we will look at the most common ones and their meanings.

The flowline



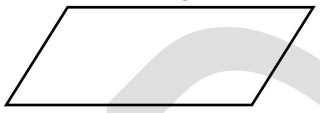
This is a flow line (an arrow). These show how the program is supposed to flow from one shape to another.

The Oval



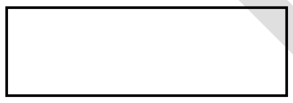
This shape is used for START and STOP. These show where the flowchart begins and ends.

The Parallelogram



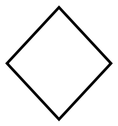
This shape is used for INPUT and OUTPUT. That is, whenever you want the user to enter a value, or you want to display a value to the user.

The Rectangle



This shape is used for processes. These processes include mathematical calculations and variable assignment (giving a value to variables).

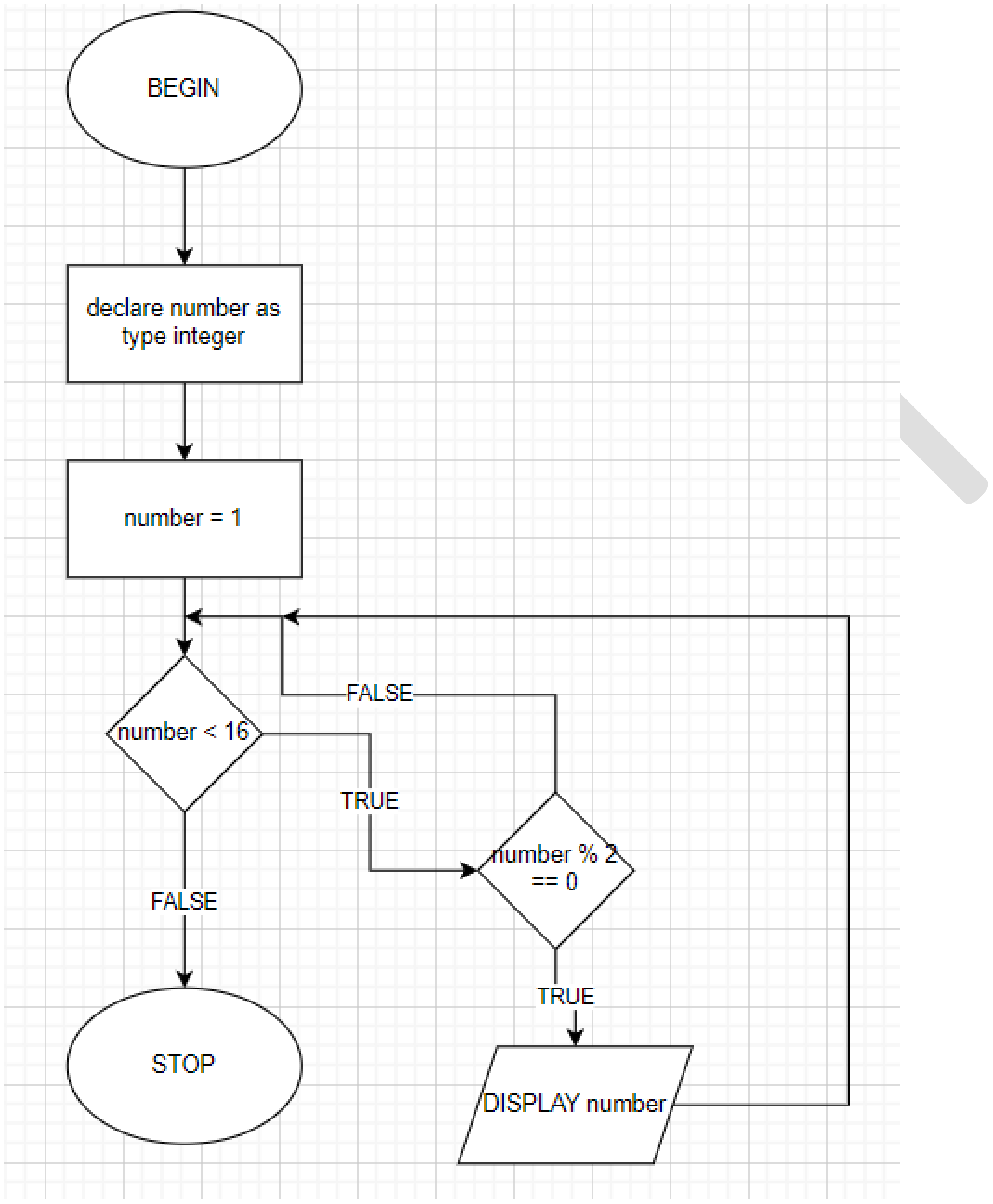
The Rhombus



The Rhombus is the shape used to evaluate a given condition, and then do 1 of 2 results depending on the condition's result. This will be easier to demonstrate by use of an example.

The Example

For this example, we will make use of the same previous problem… that is, displaying only the even numbers within the range of 1 and 15 inclusive. We will now put together all that we have learnt thus far.



End of Chapter 1

This brings us to the end of our introductory chapter.

In this chapter, we would have looked at introductions to some of the fundamentals that you will need to move forward and grow in computing programming, namely:

- Problem Solving
- Algorithms
- Narrative Writing
- Pseudocode
- Flowcharts

All of which will be discussed in detail in their respective chapters.

As it will be more practical to have an understanding about syntax for Pseudocode, in the next chapter, we will focus on that, before moving onto practicing Problem Solving and Algorithms.

Chapter 2: Programming Fundamentals

Overview

Welcome to chapter 2. The focus of this chapter will be on Programming Fundamentals. For the examples and explanations, we will be making use of Pseudocode to get you familiar with it and its application, occasionally using C++ or Python for a visual representation. By the time you complete this chapter, you will have a good grasp on the following concepts:

- IPO
    - Input
    - Processing
    - Output
- Variables
    - What they are
    - Naming
    - Creating Variables
    - Using variables
    - Constants
- Basic Data Types
    - Integers
    - Floats / Real
    - Characters
    - Strings
    - Booleans
- Arithmetic Operators
    - Addition
    - Subtraction
    - Division
    - Multiplication
    - Modulus
- Logical Operators
    - Equality
    - Inequality
    - Logical Connectives
- Types of Programming Constructs
    - Sequential
    - Selective
    - Repetition
    - Functional

IPO (Input, Processing, Output)

As we would have discussed, programs exist for us to solve problems. What we haven't looked at however, is how exactly they do this. Well prepare yourself because that will be the focus of this section.

For a program to function, it generally has 3 things it needs. These are:

- Inputs
- Processing
- Outputs

The inputs give the program the information it needs to carry out its task.

Processing is the actual weight of the program, which processes the information it would have received as inputs, and then calculate some form of result.

Outputs are what the program will display as a result of processing. Outputs have various forms including but not limited to:

- Text on screen
- Audio
- Some action

We will look at a basic example making use of Pseudocode.

Example:

Write a program that gets a number from a user, adds 5 to it, and then displays the number.

To solve this problem, we first break it down into Inputs, Processes and Outputs. To easily do this, we will make use of an IPO chart.

INPUTS	PROCESSES	OUTPUTS
A number	Add 5 to the number	The new number

Using an IPO chart allows us to easily visualize our 3 categories of information and makes building our program easier. Now we can work on the solution.

```
BEGIN
  // Remember, this is a comment
  // First we get the input from the user, by making use of the READ keyword
  READ number
  // Now we add 5 to that number
  number = number + 5
  // Now we display the new number making use of the DISPLAY keyword.
  DISPLAY number
END
```

To get input from a user's keyboard in pseudocode as shown above, we can make use of the **READ** keyword. The format is as follows:

READ variable_name

WHERE variable_name is the name of the variable to store the received data.

To display output to the screen, we make use of the **DISPLAY** keyword. The format for this is:

DISPLAY "some text here" OR DISPLAY variable_name

WHERE "*some text here*" is text wrapped in matching single or double quotes (" " ' '), and variable_name is the name of a variable. When either of these statements are executed, the value after the **DISPLAY** keyword, will be sent to the screen for you to view.

Thus far, we have encountered the term *variable* countless times, but still have received no explanation. Now that we have completed our look at basic I/O (Input / Output), we can move on to focus on processing.

Before looking at data types however, first we bring our attention to *variables*.

Variables

When explaining what variables are, I like to relate them to boxes. A variable is like a box that you can only put one item in at a time. If you want to put in a different item, then you end up taking out the previous item, and then putting in the new one.

Our powerful boxes, variables, are part of our device's memory, that is reserved for storing a particular bit of data. There are different data types (that we will look at shortly), and therefore, each will be stored differently. For example, you wouldn't expect a simple text file to take up nearly as much storage as a mp4 file (video) will. This is because they are different file types, and represent different types of data, and have different requirements for storing.

For every one of these boxes you create, you must give it a name that you will use to find the box and get what is inside of it. As such, variable names are like labels that you will put on your box to know which is which, and what should have in what. In programming, these variable names have rules that they have to follow to be considered valid.

Variable names

Variable names must generally follow the following rules to be considered as valid:

1. Variable names must not contain spaces. Use underscores (_) instead.
2. Variable names must not begin with a number but may include a number anywhere else in its name.
3. Variable names must not contain any symbol (!@#$%^&*()-+={}) etc. The only exception being the underscore.
4. Variable names may begin with an underscore.

5. Variable names are generally case sensitive. That is, ***myVariable***, ***myvariable*** and ***MyVariable***, refer to three (3) different variables.
6. Variable names **SHOULD** be clear and describe the value the variable is meant to have. This is recommended for readability purposes.

When naming variables, you must also consider not just yourself, but also any future developer that may have to work with your code, including your future self. Remember, what may make sense to you in the moment, may be completely nonsensical to you in the future, or to anyone else. I say this from experience, as having sillily named a bunch of variables stuff like, ***yes***, ***no***, ***maybe*** and ***thing***. Needless to say, when I had to go back to make fixes, I ended up having to rewrite that entire section of code because I could not figure out what anything did 🤣.

Creating a variable

Now that we know the rules for naming our powerful boxes, let's find out how to make them.

The way to make a variable depends on how the programming language is typed. There are two (2) main types, these being Static and Dynamic.

Static typed languages require that variables be declared before they are used. Declaration simply means that we tell our program that we want to create a variable x, which has the type of y.

In a static typed language, when we create a variable, we are telling our program to reserve space in the computer's memory of a specific size to store a certain type of value, and then label that reserved space with our variable_name, so we can get access to it.

In pseudocode, we make use of this static approach for declaring our variables. To declare a variable in pseudocode we use the following statement:

DECLARE variable_name AS TYPE data_type

Where **variable_name** is the name of the variable and **data_type** is the type of data the variable will contain.

In a dynamically typed language, there is no need to reserve the space for a variable beforehand. We can simply create and use them as we please. Whereas in a Static typed language, once a variable is declared with a specific data type, it can only hold data of that same data_type, in a Dynamically typed language, the type of a variable is dependent only on the type of the data it is holding and will change depending on the data.

This information may not seem relevant now, but it will become important when you graduate from pseudocode and choose a programming language to use.

Now as a recap, we are going to create variables called *number* and *letter*, which will have the datatypes *integer* and *character,* respectively.

```
BEGIN

  // Integer is a numeric data type that represents all whole numbers
  // Whole numbers are numbers without a decimal point, both positive and negative
  Declare number as type Integer

```

```
    // Character is a data type that represents any single character.
    // Example: 'a', 'b', '!', '@', '&'
    Declare letter as type Character

END
```

We will look at datatypes more in an upcoming section.

Now that we know how to create a variable, now we can look into giving them values.

Assigning a value to a variable

To assign a value to a variable, we make use of the *assignment* operator (=) using the format:

variable_name = value

Where the name of the variable that we want to store data in is on the left side of the equal sign, and the value that we wish to store inside the variable, is on the right side.

This action of giving a variable a value is referred to as *assigning*.

When assigning a value to a variable in pseudocode, we may either use the above approach or:

variable_name <- value

Which is read as: "variable_name is assigned value" or "value is assigned to variable_name"

Either approach is fine in pseudocode, but to keep it similar to what you will encounter when you pick up a programming language, I will continue to make use of the assignment operator (=).

For a "techier" view, when you use that assignment statement, you are telling your program to take the value, and store it in the memory location under the name *variable_name*, so that when you call on *variable_name*, it knows to give you that value.

Using the box analogy, consider it as you tell your box manager, to take your book put it in a box in your basement called *myBook*. Now when you ask your box manager to bring you the item in the box labelled *myBook*, then he will know to return your book to you.

Using a variable

To use a variable, we simply refer to it by the name we gave it. With this name, we can modify the contents of the variable, or simply retrieve them for whatever purpose we need, such as displaying it to the screen. Example:

```
BEGIN
  // Here we declare a variable called number, to be the type of integer
  Declare number as type Integer
  // Here we assign the value 5, to the variable named number
  number = 5
  // This will display the contents of number (in this case 5) to the screen
  DISPLAY number
END
```

As we continue to progress through the book, we will continue to get more familiar with the concepts of variables and their uses. Before we move onto data types, I want to introduce the concepts of Constants briefly.

Note: A variable may only hold data corresponding to the type that the variable was declared with. That is, a variable declared with the data type of integer, may only hold integers, and one declared with the data type of real, may only hold real numbers.

Constants

We know that variables are like containers / boxes that we can create, then modify the contents as we wish. Constants are similar to variables, in the sense that we can create them to store data, that we can then retrieve at any time by calling its name. The difference lies in modification. Whereas variables can change their value at any time by use of the assignment operator (**=**), the value of a constant remains the same as when it was declared and cannot be modified after. Therefore, it is constant. An example of a constant would be PI. The mathematical constant PI has a set value, which cannot be changed, but can be used continuously in our calculations.

To declare a constant in pseudocode, we use this statement:

DECLARE CONSTANT variable_name AS TYPE data_type WITH VALUE value

Where variable_name is the name of the constant, data_type is the type of data the constant is expected to hold, and value is the data that the constant would be expected to hold.

An alternative to the above is replacing ***CONSTANT*** with the shortened form ***CONST***.

Practice: Create an integer constant named *myConstant* with the value of 10

```
BEGIN
  Declare const myConstant as type Integer with value 10
END
```

With constants covered, we can finally look at these datatype things we've been mentioning so much recently.

Basic Data Types

In real life, there are different types of data. Integers, real numbers, and text just to name a few. Depending on the type of data, the things we can do to it differ. For example, you wouldn't be able to divide the word "dog" by the word "cat", nor could we add the number 5, and the word "house".

To represent these various types of data in programs, most programming languages implement what are known as data types. In this section, we will look at some of the most basic ones that you will most likely be using whenever you decide to code up a new project.

These are:

- Integers
- Real numbers
- Characters
- Strings
- Booleans

Integers

As you may recognize from math, Integers are whole numbers (numbers with no fractional part), that may be either positive or negative. Examples of integers include: 54, -123, 20134, -123124, 0, 1, -1 and 21.

To declare a variable to have the datatype of integer, we use the following statement:

DECLARE variable_name AS TYPE integer

Where variable_name is the name of the variable to be declared as an integer.

Example: Create a variable named myFirstInteger and give it the datatype of integer.

```
BEGIN
  Declare myFirstInteger as type Integer
END
```

Floats / Real

Floats or real numbers are any number that can be represented as a fraction or contains a fractional part. In programming, for a number to be considered as a real number, then it MUST contain a decimal point. That is to say, 10 and 10.0 are not of the same time. 10 will be recognized as an integer, whereas 10.0 will be recognized as a real number, despite essentially containing the same value. However, when comparing the 2 using code, that is, checking to see if 10 is equal to 10.0, the result will be True.

Note: Some programming languages may refer to real numbers as floats (floating point numbers)

To declare a variable as a real number in Pseudocode, we use the following format:

DECLARE variable_name AS TYPE real

Where variable_name is the name of the variable whose data type is being set to real.

Example: Create a variable named myRealNumber of the data type Real and give it the value of 7.5.

```
BEGIN
  Declare myRealNumber as type Real
  myRealNumber = 7.5
END
```

Characters

The character data type is used to represent a single character. These are generally represented by the character wrapped in single quotes. Examples: ‘1’, ‘5’, ‘a’, ‘A’, ‘@’, ‘*’, ‘^’.

When declaring a variable to have the data type of character, we use the following format of:

DECLARE variable_name AS TYPE char

Where as usual, variable_name is the name of our variable that we want to be of the character data type.

Example: Create a character typed variable with the name myFirstChar and store the character ‘+’ inside of it.

```
BEGIN
  Declare myFirstChar as type Char
  myFirstChar <- ‘+’
END
```

Strings

Strings are a common data type. They are essentially a collection of characters, wrapped in double quotes. Examples: “cat”, “monkey”, “car”, “some other really nice sentence”.

Example: Declare a variable named myNewString to hold the string “Pseudocode is a fundamental”.

```
BEGIN
  Declare myNewString as type String
  myNewString = “Pseudocode is a fundamental”
END

```

Booleans

The last of the basic data types we will look at are Booleans.

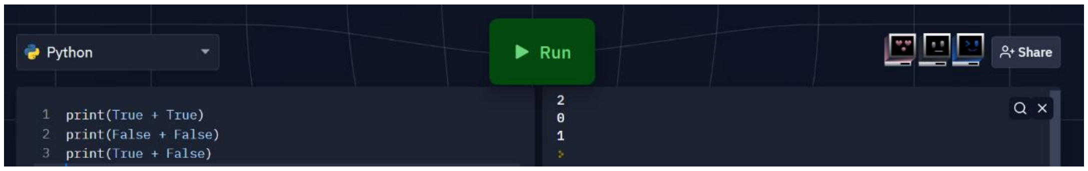
Booleans are an extremely simple data type, capable of only storing 2 values, these being True and False.

Making use of Booleans, we can make use of logic to evaluate conditions. To create a Boolean in pseudocode, we use the following statement.

DECLARE myBoolean as type Bool

Although you may not directly use this data type, your program will still use it implicitly, such as when using If statements and evaluating conditions.

Booleans are generally implicitly used as 1 and 0, which represent True or False respectively. As such, despite us using the terms True and False, what I am about to show you in this next image is possible:



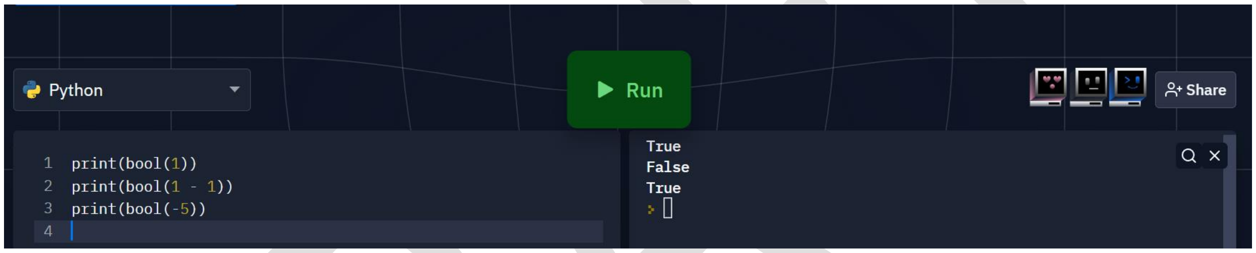
```
print(True + True)
print(False + False)
print(True + False)
```

```
2
0
1
```

Figure 2: image showing addition of Booleans

You may initially be confused as to how we are adding values like True and False. Well, remember that I said that Booleans are implicitly used as 1 and 0. The best way to think of it, is that True and False are actually constants, holding 1 and 0 respectively. As such, True + True, is saying 1 + 1. False + False is saying 0 + 0. Lastly, True + False, is saying 1 + 0.

I say this to bring up an important point. Any value that evaluates to 0, will be False, including the number 0. Any other non-zero value, negative numbers included, will evaluate to True, as shown in the below image.



```
print(bool(1))
print(bool(1 - 1))
print(bool(-5))
```

```
True
False
True
```

Figure 3: Image showing bool values

Note: The *bool(value)* I have in the above image, just means to take whatever value is, and evaluate it to be either True or False depending on the value. 1 – 1 is 0, therefore it is False.

Arithmetic Operators

In programming, we make use of arithmetic operators to execute certain actions on data. The ones that we are going to look at in this chapter, are those that you should be well acquainted with basic math classes. In programming, depending on the type of data being operated on (the operands), the operators may behave differently.

Addition (+)

Between two numeric values (integers or floats/real numbers), the addition operator is used to add the values together, and then return a result. Examples:

```
BEGIN
  // 15 Will be displayed to the screen
  Display 5 + 10
  // 15.5 will be displayed to the screen
  Display 5 + 10.5
  // 20 will be displayed to the screen
  Display 9.5 + 10.5
END
```

Note: When adding 2 numeric data types, if they are not the same, then conversion takes place implicitly towards the one with higher priority. In the case of addition between an integer and a real number, the result will be a real number. If you try to store the result of this addition in a variable of the integer data type, then the fraction part of the number will be returned, that is, only the whole number part will remain. Example:

```
BEGIN
  Declare myInt as type Integer
  // As we know, 10 + 10.5 gives us an answer of 20.5
  myInt = 10 + 10.5
  // Our output, however, will be 20.
  // As integers do not know how to handle fractional parts
  Display myInt
END
```

However, when used between 2 strings, concatenation takes place. That is, the string on the right side of the + sign, will be added to the end of the string on the left side of the + sign. Example:

```
BEGIN
  Declare myString as type String
  myString = "Cat" + "Dog"
  // CatDog will be displayed to the screen
  Display myString
END
```

Subtraction (-)

Subtraction may only be done between 2 numeric values. Similar to addition, in the case of subtraction between an integer and a real number, the result will by default be a real number. As such, if stored in a variable with the data type of integer, the fractional part of the number will be lost, and only the whole number will remain.

Division

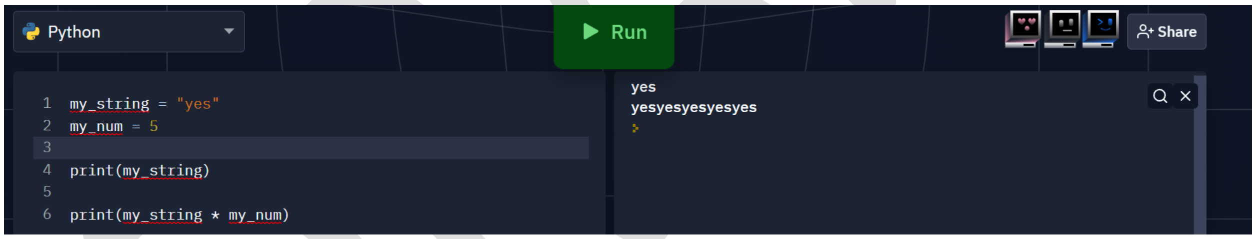
Division may also only occur between two numeric values. It will be wise to note that division will ALWAYS return a float, regardless of the types of the operands, as the result of the division. Example

```
BEGIN
  // Division between 2 integers
  DISPLAY 10 / 5      // Displays 2.0, a float
  // Division between 2 floats
  DISPLAY 6.6 / 3.3   // Displays 2.0
  // Division between a float and an int
  DISPLAY 6.0 / 3     // Displays 2.0
END
```

Multiplication

When used between two numeric values, the product of the multiplication will depend on the operands used. If a float / real number was involved during the multiplication, then the result will be a real number. However, if it is a multiplication only involving 2 integers, then the result will be an integer. Example:

In Python, my main programming language, multiplication has another function, and that is with a string and an integer as operands. Multiplication between a string (x) and an integer (y), will cause the string x to be duplicated y times. Example:



```
my_string = "yes"
my_num = 5

print(my_string)

print(my_string * my_num)
```

```
yes
yesyesyesyesyes
```

Figure 4: Python code to show string replication

You can use that same website here to try out the code yourself if you want. I do apologize for the red underlines under variable names, that's just my browser telling me those aren't proper words 😅.

Modulus

Modulus is a type of division, that only returns the remainder of the division. For example, 10 / 2, would be 5 remainder 0. Therefore, 10 % 2, will give us 0. Another example would be 11 / 3. This will give us 3 remainder 2. Therefore, 11 % 3 will give us 2.

In an earlier example, when we were asked to find a number that was even, you may recall that we had the following code:

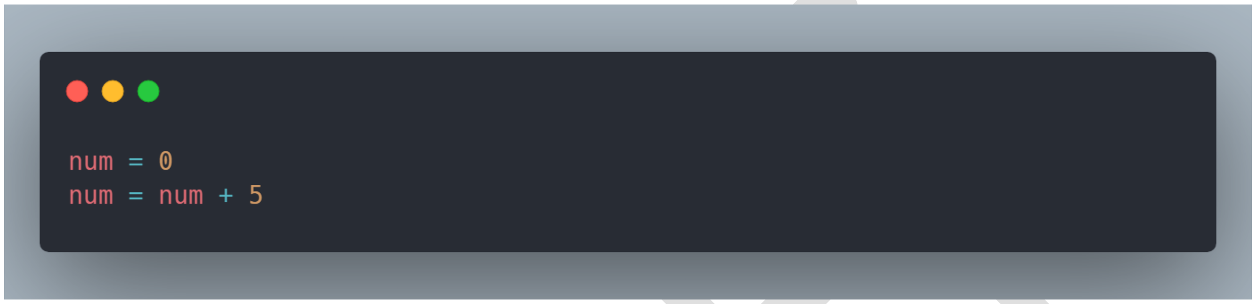
If (num % 2 == 0) then

We will look at if statements shortly, but right now, the focus is on what is inside of the brackets. The reason why we used modulus here, is because rightfully, for a number to be even, it has to be completely divisible by 2. That is, when divided by 2, there must be no remainder. What we essentially do there, is check if the value of *num* has a remainder when divided by 2. If the remainder is 0, then *num* has in an even number. Otherwise, it is an odd number.

The section of code I am talking about can be found here.

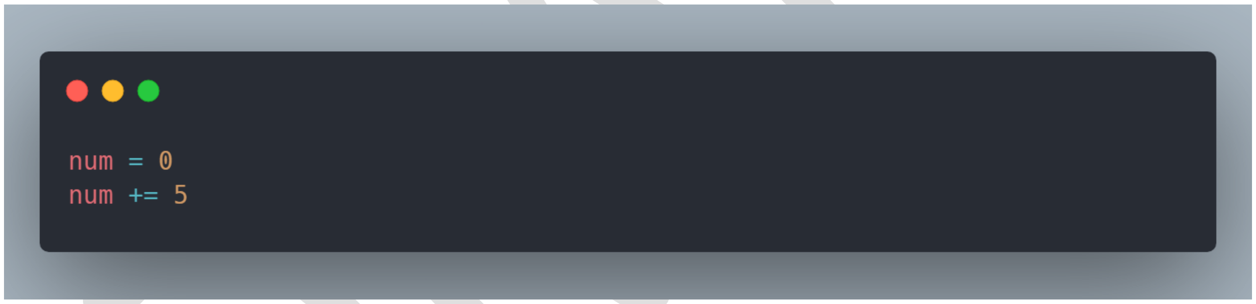
In Place Assignment

Earlier we looked at assigning a value to a variable. Now that we have looked at the basic arithmetic operators, now is a good time to bring up In Place assignment operators. These operators combine the effects of an arithmetic operator (+-*/%) with that of the assignment operator to get both effects. Take a look at the example snippet below:



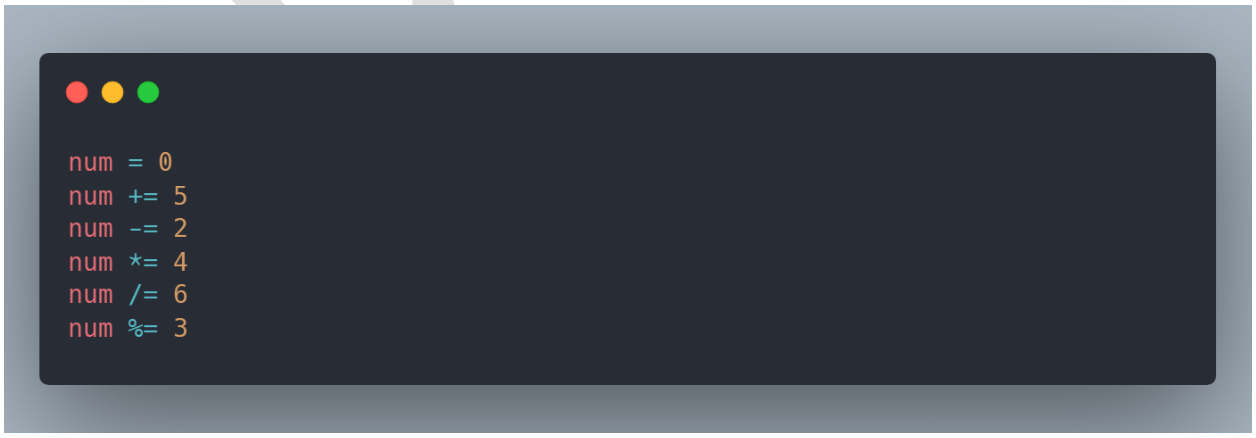
```
num = 0
num = num + 5
```

This is done by making use of the assignment operator, to assign the value of num + 5 to the variable of num. However, this can be done simpler by way of an in-place operator. I.e.



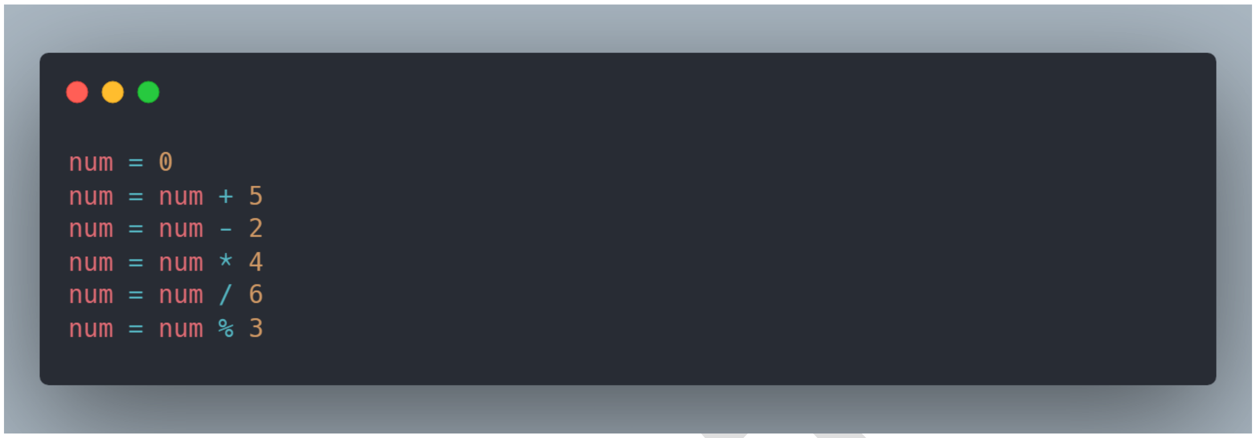
```
num = 0
num += 5
```

These 2 snippets of code are identical in functionality. The main difference is that it removes the redundancy from having to repeat the name of the variable when you are making an arithmetic change to it. These in place operators exist for every arithmetic operator. For example.



```
num = 0
num += 5
num -= 2
num *= 4
num /= 6
num %= 3
```

This above snippet functions identically to the below, expanded snippet.



```
num = 0
num = num + 5
num = num - 2
num = num * 4
num = num / 6
num = num % 3
```

Order of Operations (Arithmetic Operators)

When multiple operators are in the same line to evaluate some condition, there is an order as to which operation takes place first.

For example, given the equation: 10 + 3 * 5, what do you think the answer will be.

If you said 25, then you would be correct. But, why?

This is because multiplication has a higher priority than addition, and therefore will be completed first. However, take a look at this next situation.

Given the equation: (10 + 3) * 5, what do you think the answer will be?

If you thought 65, then +10 points for you. But, why?

This is because brackets have the highest priority and will be evaluated before anything else.

Here's a table of the order of precedence for your reviewal in order of decreasing priority. As we meet more operators, this list of ours will expand.

Operator	Description
()	Brackets
/ * %	Division, Multiplication, Modulus
+ -	Addition, Subtraction

In cases where operators on the same level are in the same statement, then the operations will be evaluated from left to right. Example:

Given the equation: 10 * 2 / 5, what do you think the answer will be?

The correct answer will be 4.0. Since * and / are on the same priority level, the multiplication, which is on the leftmost side, will be evaluated first, then the division.

That brings us to the end of the basic Arithmetic Operators. Now, we turn our attention to Logic Operators.

Relational Operators

From mathematics, you should know of the 6 basic relational operators, them being: ==, !=, <, <=, > and >=. In programming, they have the same function as they have in mathematics and will evaluate to True or False depending on their operands.

The first one we will be looking at is the **==** operator and can be read as "is equal to".

Example:

```
BEGIN
  DISPLAY 5 == 5    // Will display True
  DISPLAY 10 == 10.0  // Will display True
  DISPLAY 10 == (5 * 2)         // Will display True
  DISPLAY 2 == 3      // Will display False
END

```

This operator gives a result of True if the statement is True and gives a result of False otherwise.

Now, for the **!=** or **<>** operator, both of which are read as "is not equal to". This operator returns True if the operands are unequal, and False if they are equal.

Note: Depending on your teacher, you may use **<>** instead of **!=**, so just note that fact. However, both of them serve the same purpose

Example:

```
BEGIN
  DISPLAY 5 != 5 // Displays False
  DISPAY 10 <> 2 // Displays True
END
```

Example:

```
BEGIN
  DISPLAY 5 < 2 // This will display False, as 5 is not less than 2
  DISPLAY 5 > 2 // This will be True, as 5 is more than 2
  DISPLAY 5 >= 5 // This will be True because although 5 is not more than 5, it is equal
  DISPLAY 5 < 5 // This is False as 5 is not less than 5.
END

```

Logical Operators

We know that arithmetic operators are used to evaluate some mathematical operation, such as addition or multiplication and that relational operators are used to compare values looking for equalities or inequalities. Logical operators on the other hand, are used to create and evaluate a complex condition.

There are three of these Logical operators and they are:

1) AND
2) OR
3) NOT

Of these three logical operators, two are binary, as in they require two operands (a left and right value) for them to work. It is these two (AND and OR) that we will be focusing on for this section. The NOT operator will be looked at in the next section as it is unary.

AND and OR are known as logical connectives, as they are used to connect logical expressions and returns a single truth value based on their operands.

Starting with ***AND*** consider the following statement:

If I am tired AND it is nighttime, then I will go to sleep.

Evaluating this is simple. All this means is that for me to go to sleep, I have to be tired, **AND** it has to be nighttime. If I am tired but it is not nighttime, or it is nighttime, but I am not tired, or I am neither tired, nor it is nighttime, then I will not go to sleep.

Thus, looking at AND, we can conclude that for a statement to be evaluated as True, both operands have to evaluate to True. Here are some examples in code.

```
BEGIN
  Display True AND True // Will display True
  Display True AND False // Will display False
  Display False AND False // Will also display False
END

```

This can be represented in the following truth table.

Operand 1	AND	Operand 2	Result
TRUE	AND	TRUE	TRUE
TRUE	AND	FALSE	FALSE
FALSE	AND	TRUE	FALSE
FALSE	AND	FALSE	FALSE

Next is OR. Consider the following example:

If I am tired OR it is nighttime then I will go to sleep

What this means is that if either of these conditions are True, that is, if I am tired, or if it is nighttime, then I will go to sleep. The only time I will not go to sleep, is if I am neither tired, nor it is nighttime.

This is represented in the following code

```
BEGIN
  Display True OR True // Will display True
  Display True OR False // Will also display True
  Display False OR False // Will display False
END
```

Now we represent this in the below truth table.

Operand 1	OR	Operand 2	Result
TRUE	OR	TRUE	TRUE
TRUE	OR	FALSE	TRUE
FALSE	OR	TRUE	TRUE
FALSE	OR	FALSE	FALSE

Note: Most programming languages tend to represent logical ***AND*** and ***OR*** as **&&** and **||** respectively. However, for easier readability with pseudocode, we will continue to use ***AND*** and ***OR***.

Unary Operators

Thus far, all the operators that we have looked at have been binary. Not binary as in 0s and 1s, but binary as in 2, that is, they required 2 operands (values, one on both sides of the operator) to function. Now, we are going to review Unary operators. As the name implies, these operators only require one operand to function.

Unary + and –

First on the list are the unary version of + and -, called unary plus and unary minus respectively. These are called instead of addition and subtraction, when they are provided only one operand, that is to the right of the operator.

Examples: +5, -2, +10, -9

Seeing the examples above, I am certain that you were able to understand their purpose. If you thought that they were used to declare a number as positive or negative, then you were correct 😁.

Unary ++ and --

Some programming languages make use of these 2 unary operators, which are known as the increment and decrement operators.

These are used to affect the value of a variable containing a number, by 1. Increasing the value by 1 for the increment operator and decreasing the value by 1 for the decrement operator.

There are 2 variations of these operators, known as pre and post.

Pre-increment and pre-decrement, involve having the operator placed right before the variable. Example:

```
BEGIN
  Declare num as type Integer
  num = 5
  Display ++num // Will display 6
  // num now has the value of 6
  Display --num // Will now display 5
  // num now has the value of 5
END
```

Post-increment and post-decrement, however, involve the operator being placed directly after the numeric variable.

Example:

```
BEGIN
  Declare num as type Integer
  num = 4
  Display num++ // Will display 4. Will explain after
  // At this point, num now has the value of 5
  Display num-- // Will display 5. Will explain after
  // At this point, num now has the value of 4
END
```

At this point, you may have started to realize that we are getting different results when we display the value to the screen depending on whether we use the pre or post forms of the ++ and -- operators.

The difference between the 2 forms, is in the order of which they are done.

For the pre versions, notice how the ++ and -- come before the actual variable name? This means the operation (the addition or subtraction) is actually done before the value of the variable is displayed.

However, for the post versions, notice how the ++ and -- come after the actual variable name? This means that first the value of the variable is provided to be displayed, and THEN the operation (addition or subtraction) is done.

Unary not (!)

Last but *not* least, is the unary !, read as “not”. This is a logical operator used to negate the current truth value of an expression that it precedes. That is to say, any expression that was True, will become False, and any expression that was False, will now become true.

Example:

```
BEGIN
  DISPLAY !(5 == 5) // 5 == 5 is True, so this will display False
  DISPLAY !(5 != 5) // 5 != 5 is False, so this will display True.
  DISPLAY !(5 > 10) // 5 > 10 is False, so this will display True
END

```

With all of the basic operators covered, we can now update our precedence table. The below operators are listed in decreasing priority (from highest priority to lowest).

Operator	Description
()	Parentheses (brackets)
++ --	Post increment Post decrement
+ - ! ++ --	Unary plus Unary minus Unary logical NOT Unary pre-increment Unary pre-decrement
/ * %	Multiplicative
+ -	Additive
< <= > >=	Relational
== !=	Equality
and (&&)	Logical AND
or (||)	Logical OR

Types of Programming Constructs

In our world, we have several different types of problems, and as such, we also have several solutions for the same problem. Also, a solution may be able to solve a problem, but it may do so in an inefficient manner which may not be ideal depending on the situation.

In this section, we will be looking at 3 types of programming constructs. Constructs, in terms of programming, refers to formats that can control the order that our programming statements are executed. We will begin with the most basic one, Sequential

Sequential Programming

This is the most basic form of programming, and simply means that the code is executed line by line, sequentially. That is, one after the other in the order that they were defined. For example:

```
DISPLAY "Hello"
DISPLAY "There"
x = 10
DISPLAY x

```

If we were to somehow run this, we will get the following output

```
Hello
There
10
```

As you can see, each line would be executed one after the other, i.e., sequentially. That is all there is to this, so we can move onto the next section.

Selective Programming

The next form of programming we will be looking at is Conditional / Selective. This makes use of **IF** statements, which determine whether a section of code is to be executed or not, based on a condition, hence the term Conditional.

IF statements

The basic format for an **IF** statement is:

IF condition THEN action

Where the condition is a statement, that is either True or False, and action is the code to be run if the condition is True.

Thinking of it in simple terms, imagine when we tell ourselves that:

IF it is raining THEN I will walk with an umbrella.

This follows the same IF condition THEN action syntax and is easily understood. Another example:

IF he is hungry THEN he will eat food.

With some basic English examples, now we can try doing it in code.

Example:

```
IF (1 == 1) THEN
  DISPLAY "One is equal to One"
ENDIF
DISPLAY "Above us is an if statement"
```

If we were to run this, THEN our output would be:

```
One is equal to One
Above us is an if statement
```

But, what happens when the condition is false. For example:

```
IF (2 == 1) THEN
  DISPLAY "2 is equal to 1"
ENDIF
DISPLAY "Above us is an if statement"

```

If we were to run this pseudocode, our output would be:

```
Above us is an if statement
```

This shows us another thing about how the IF statement works.

The IF statement will execute its code ONLY IF the condition is True. However, regardless of whether the condition is True or not, program execution will continue on the next line that is not a part of the IF statement. This explains why the line ***"Above us is an if statement"*** was displayed both times.

So, we can use the IF statement to check a condition, and then decide to do an action depending on if the condition is True. But, what about when the condition is False? Will we have to make a new IF statement to check for it?

Luckily, we don't have to. Introducing the ELSE statement.

The ELSE and ELSE IF statements

The ELSE statement is an extension of our IF statement, which will ONLY be executed, if the condition given to the IF statement is False.

The ELSE statement has the format: ELSE action.

Putting this together with the IF statement, we get:

IF condition THEN action ELSE action

Let's try an example in plain English.

IF outside is raining THEN we will stay home. ELSE, we will go to the beach

Another example:

IF he is hungry THEN he will eat food. ELSE, he will go to sleep

Now let's try doing this in code using our same 2 previous examples.

```
IF (1 == 1) THEN
  DISPLAY "The numbers are the same"
ELSE
  DISPLAY  "The numbers are different"
ENDIF

```

Executing this will give us the following output:

The numbers are the same

Now let's try it with the second example.

```
IF (2 == 1) THEN
  DISPLAY "2 is equal to 1?"
ELSE
  DISPLAY "2 is not equal to 1"
ENDIF

```

With this, our output will be:

2 is not equal to 1

With an IF THEN ELSE statement, it's either only the IF action or the ELSE action that can be executed. Not both.

This is not all the Selective Construct has for us. What if, we wanted to check for several conditions, and execute different sections of code depending on the results? Introducing the ELSE IF statement.

This essentially takes the ELSE statement and allows it to be used with an IF statement, allowing you to check for a different condition without having to make a whole new IF statement.

Example in English:

IF it is raining THEN we will stay inside. ELSE, IF it is hot THEN we will go to the beach. ELSE we can go to the park.

Now we can try an example using pseudocode.

```
IF (1 > 2) THEN
  DISPLAY "1 is greater than 2."
ELSE IF (1 < 2) THEN
  DISPLAY "1 is less than 2"
ELSE
  DISPLAY "1 is equal to 2"
ENDIF
```

Running this, our output will be from the first statement whose condition is True. Therefore:

1 is less than 2

Review

Some things to note:

- You may have as many ELSE IF statements as you want.
- If using ELSE IF statements, your ELSE statement should always be AFTER all other statements.
- The ELSE statement can be thought of as the "Catch all" and will respond to any and every case that is not handled by the previous IF or ELSE IF statements.

This concludes this section on selective programming, allowing us to move onto the next type of programming constructs, that being, Repetitive programming.

Repetitive Programming

This construct focuses on using loops to repeat sections of code until a stop condition is reached. We will be looking at 3 forms of these loops, namely the **FOR** loop, the **WHILE** loop and an honorary mention, the **DO…WHILE** loop.

The For Loop

The **FOR** loop is a powerful tool, which allows us to repeat a section of code for a **SPECIFIED** number of times.

We use the for loop when we want to repeat a section of code for a known number of times (emphasis). Curious as to why we would ever want to repeat code? Well, here are some examples done in Pseudocode.

Example 1:
Say that we wanted to display numbers 1 through 10 inclusive in our program. This is how we would normally attempt to do it.

```
BEGIN
  Display 1
  Display 2
  Display 3
  Display 4
  Display 5
  Display 6
  Display 7
  Display 8
  Display 9
  Display 10
END
```

Yes, this method works, but what if we had to write a program to display the ID of each student in the school, and there were over 1000 students. Would we really have to write over 1000 lines to display them all? Well, thankfully, it doesn't have to be that way. Introducing For loops.

To use a For loop in pseudocode, we use the following structure:

```
FOR variable_name = start_value TO end_condition DO
  // Code to be repeated can go in here
ENDFOR
```

Where variable_name is the name of the integer variable that we want to use for the For loop, start_value is the value we want our for loop to start at, and end_condition is the condition that we will use to stop our For loop. The For loop will generally run until the end_condition is False.

Let's show this in an example, solving our previous problem of counting from 1 to 10.

```
BEGIN
  Declare counter as type Integer
  For counter = 1 To counter <= 10 Do
            Display counter
  EndFor
END
```

Using this same code, we can even solve our problem of needing to count from 1 to 1000.

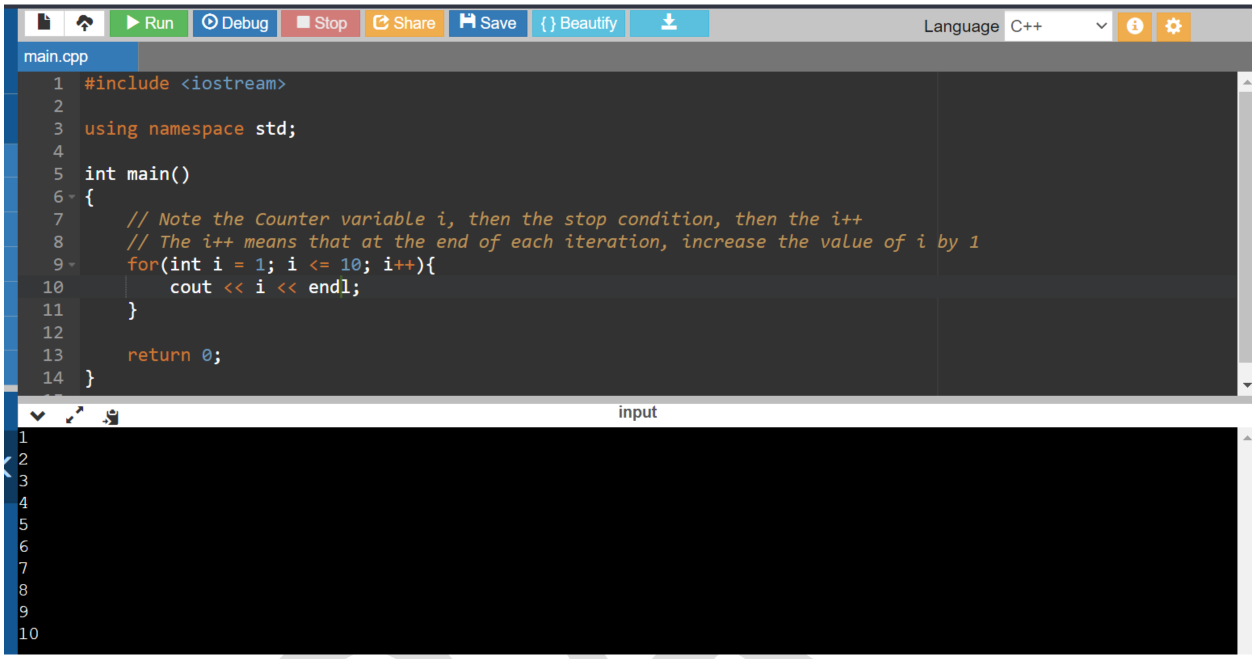
```
BEGIN
  Declare counter as type Integer
  For counter = 1 To counter <= 1000 Do
            Display counter
  EndFor
End

```

We can go all the way up to 1 million and beyond if we had to. All while using the same 3 lines that make up our For loop, and only changing our end_condition. This alone gives a clear indication of the kind of power you have with a For loop. But… How does it work?

Our counter variable acts as a tracker, holding each successive value in the loop, allowing us to use it as we please. As long as we don't actually change the value of counter, the loop will run as we expect it to.

This process of going through values using a For loop, is known as iterating, and each successful run of the For loop, is referred to as an iteration.

In programming languages, there is generally some explicit rule as to how the counter is increased. For example:

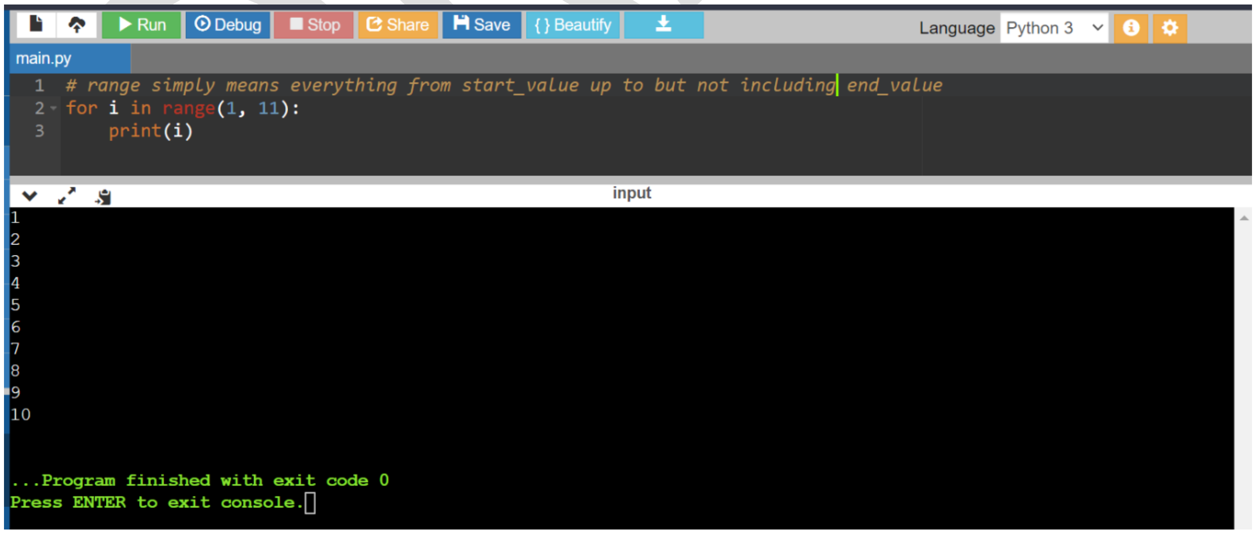


```cpp
#include <iostream>

using namespace std;

int main()
{
    // Note the Counter variable i, then the stop condition, then the i++
    // The i++ means that at the end of each iteration, increase the value of i by 1
    for(int i = 1; i <= 10; i++){
        cout << i << endl;
    }

    return 0;
}
```

```
1
2
3
4
5
6
7
8
9
10
```

Figure 5: Counting from 1 to 10 in C++

In the case of C++, the way the counter is incremented, is defined inside of the For loop. In the example above, this is by i++, which if you can recall, means to add 1 to the value of i. Another example:



```python
# range simply means everything from start_value up to but not including end_value
for i in range(1, 11):
    print(i)
```

```
1
2
3
4
5
6
7
8
9
10


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 6: Counting from 1 to 10 in Python

In the case of Python, we use *range* function to determine the range of numbers that we want to *iterate* through. It is worth noting that when using *range*, the provided numbers only go up to, but not including the end number.

Pseudocode, however, is neither a programming language, nor real code. As such, the direction of our iteration is mainly implicit and dependent on our end condition. For example, if our end condition were lesser than our starting value, then that would imply that our pseudocode wishes to count backwards.

For example, say we want to count numbers from 10 to 1. We could use the following:

```
BEGIN
  Declare counter as type Integer
  For counter = 10 To counter >= 1 DO
          Display counter
  EndFor
END

```

By looking at our condition, we can deduce that our loop is counting backwards, from 10 to 1.

With this, we can now move onto while loops.

The While Loop

A while loop is another kind of iterative structure, that we can use to repeat a block of code. What makes it different from a for loop however is its terminating condition. To do a while loop in pseudocode, we make use of the following format:

```
WHILE condition DO
  // Code to be repeated here
ENDWHILE
```

As you can see, there's no need for a counter variable. So how do we keep track and know when to end our while loop? To answer that question, it comes down to the condition that we give our while loop.

Similar to the For loop, the while loop will run until the condition that it was given becomes False. However, unlike the For loop, unless we explicitly make it so that the condition can become False, we may easily end up with an infinite loop, that is, a loop that runs on for infinity. While this may sound like a horrible thing, infinite loops do have a purpose, such as if you NEED to keep something running, for example, a game loop or a patient's life support.

For our examples using while loops, we'll first attempt to fix our problem (again), of having to count from 1 to 10 inclusive.

Note: Although a counter variable is not necessary for a while loop to function, we can still use one if we want to. Although, we will have to manage it ourselves.

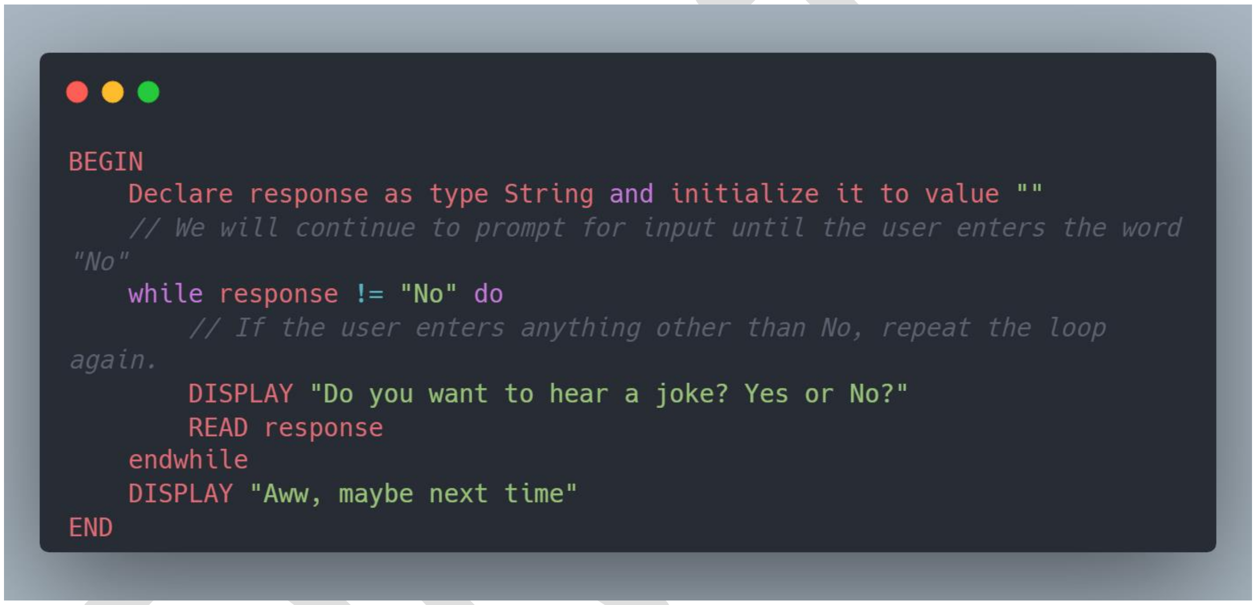
```
BEGIN
  Declare counter as type Integer and Initialize it with 1
  While counter <= 10 Do
```

```
            Display counter
            counter = counter + 1
  EndWhile
END
```

If we were to run the above code, then you will realize it gives us the same exact result as our For loop, however, requiring an extra line or 2 to do so.

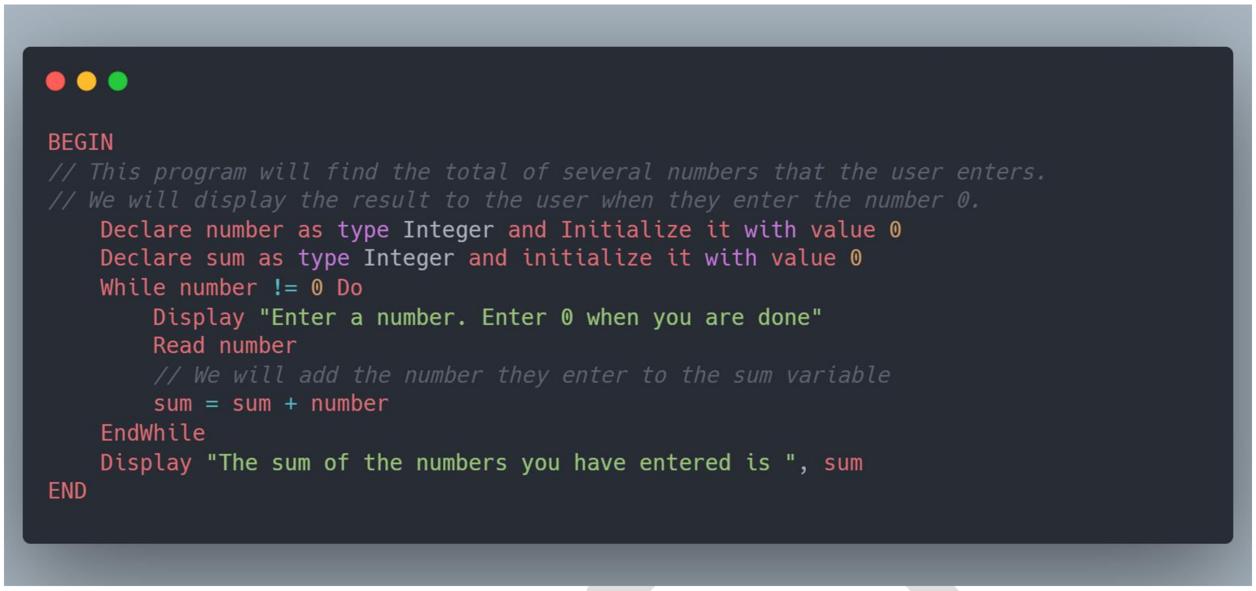
From this example, we can see that whatever we can do with a ***for*** loop, we can also do with a ***while*** loop, albeit with more steps. So, what really sets the while loop apart and makes it unique? The answer lies in the condition.

A common use for ***while*** loops is to continuously prompt for input until a desirable response is reached. Here is an example to explain.



```
BEGIN
    Declare response as type String and initialize it to value ""
    // We will continue to prompt for input until the user enters the word "No"
    while response != "No" do
        // If the user enters anything other than No, repeat the loop again.
        DISPLAY "Do you want to hear a joke? Yes or No?"
        READ response
    endwhile
    DISPLAY "Aww, maybe next time"
END
```

Of course, this example is mainly a joke, but it gives a clear indicator as to what we can do with while loops. Here is a more practical example.



```
BEGIN
// This program will find the total of several numbers that the user enters.
// We will display the result to the user when they enter the number 0.
    Declare number as type Integer and Initialize it with value 0
    Declare sum as type Integer and initialize it with value 0
    While number != 0 Do
        Display "Enter a number. Enter 0 when you are done"
        Read number
        // We will add the number they enter to the sum variable
        sum = sum + number
    EndWhile
    Display "The sum of the numbers you have entered is ", sum
END
```

In the above example, we created a simple while loop, which will find the sum of all numbers that a user enters. This while loop will continue to run until the user enters 0 as their number, at which point the loop ends, and the resulting sum is displayed.

We can make the condition of the while loop anything that we want to suit our needs, making it far more flexible than a ***for*** loop.

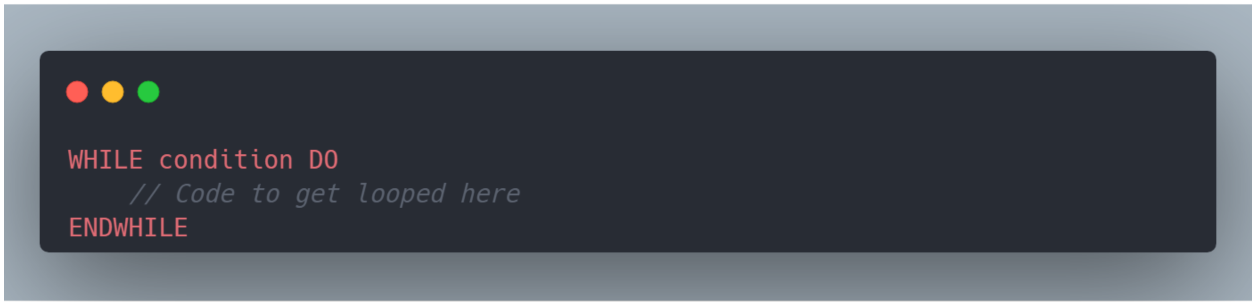
Note: You will use a ***for*** loop if you want to loop a known number of times. However, you will use a while loop if you want to repeat a section of code an unknown number of times, such as when dependent on user input.

Do While Loop

A variation of the while loop that you may come across is called the Do… While loop. This type of while loop functions identically to the regular While loop, with one exception; the code in the loop is guaranteed to run at least once. Here, let me sho

w you.

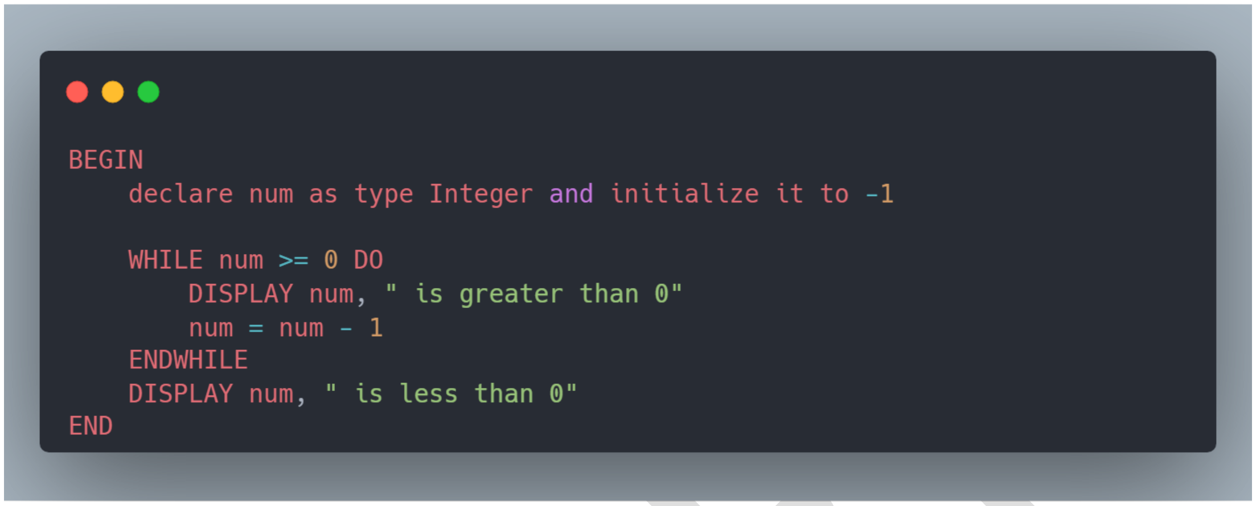
If you can recall, our regular while loop looks like this:



```
WHILE condition DO
    // Code to get looped here
ENDWHILE
```

This is fine and all, but if you look closely you may ask yourself one question. What happens if the condition is false before the while loop begins? Well, the answer to that, would be absolutely nothing.

For example, take this code snippet.

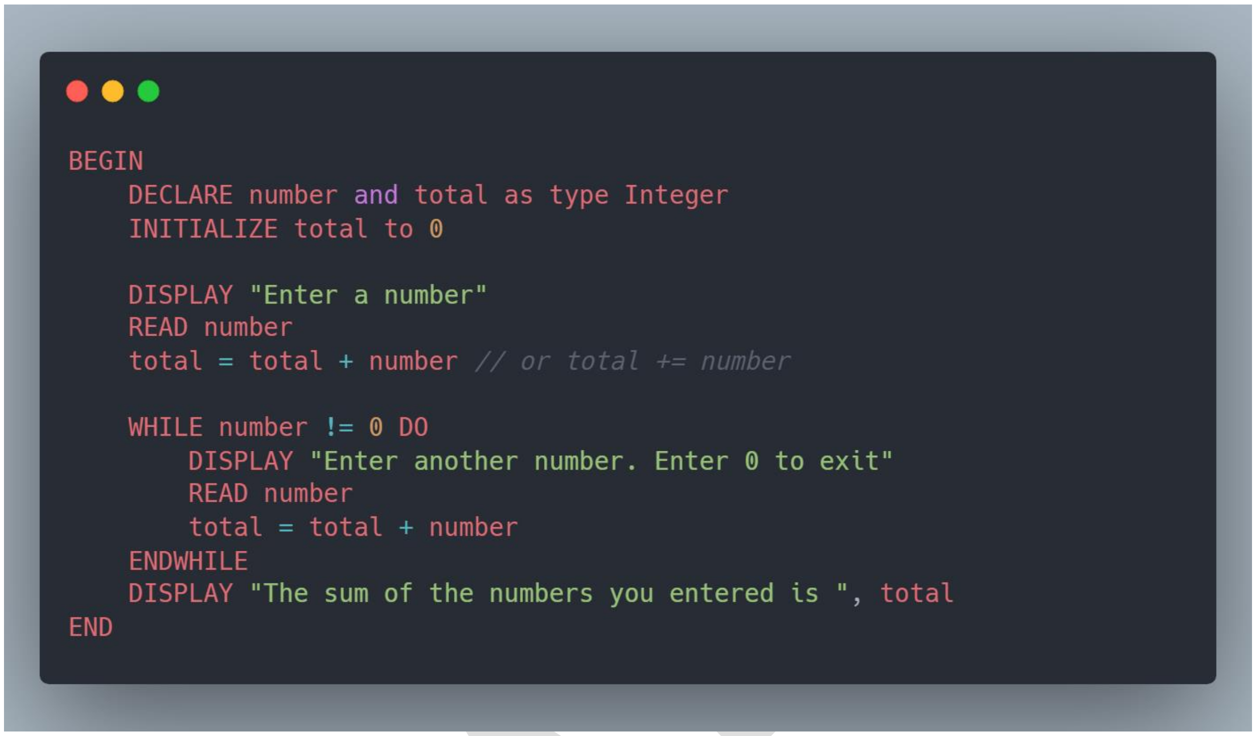


```
BEGIN
    declare num as type Integer and initialize it to -1

    WHILE num >= 0 DO
        DISPLAY num, " is greater than 0"
        num = num - 1
    ENDWHILE
    DISPLAY num, " is less than 0"
END
```

Can you tell what will be outputted when we run our program?

If you said: ***-1 is less than 0***, then I will have to give you +30 points as you are 100% correct. As we give our variable **num** a value of -1, and our while loop condition states that we only want to run the code inside of it as long as the variable **num** is greater than or equal to 0, then our value of -1 immediately makes the while loop condition false and invalid. As such, the loop is never run.

However, what if for some reason, we needed to make sure that this loop was run at least once. In this case, you may see something like this snippet here:

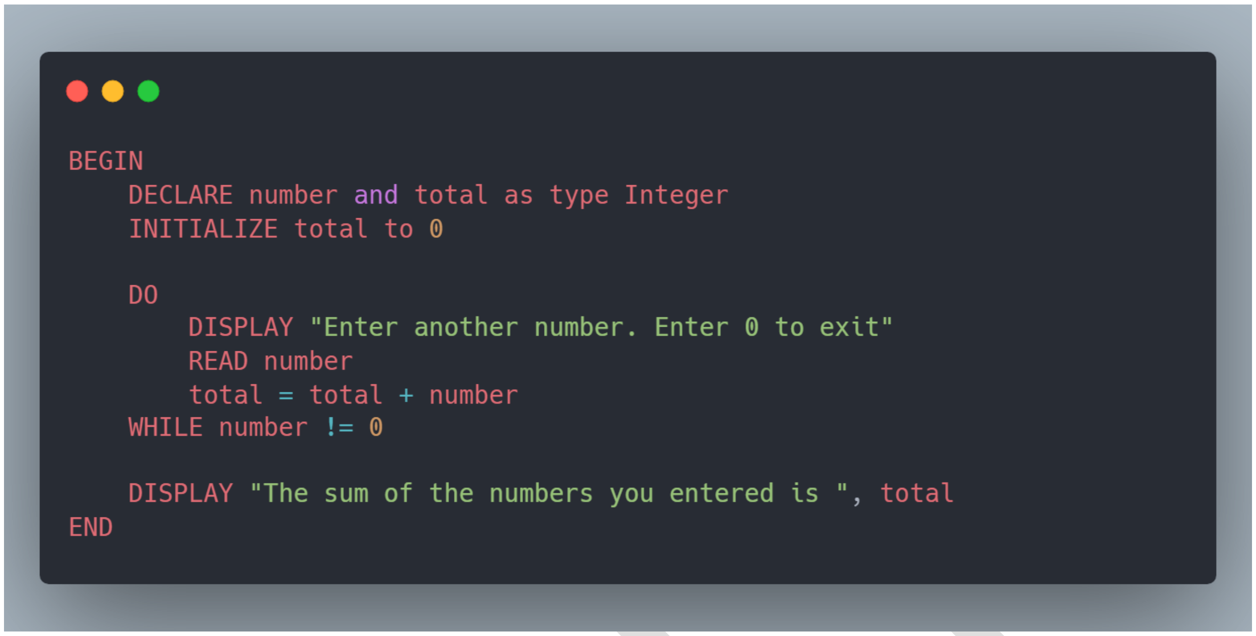


```
BEGIN
    DECLARE number and total as type Integer
    INITIALIZE total to 0

    DISPLAY "Enter a number"
    READ number
    total = total + number // or total += number

    WHILE number != 0 DO
        DISPLAY "Enter another number. Enter 0 to exit"
        READ number
        total = total + number
    ENDWHILE
    DISPLAY "The sum of the numbers you entered is ", total
END
```

Here, what we did is ask the user to first enter a number, any number. We don't care what the number is, we just want a number to add to our total. If the number the user entered the first time was 0, then we move past the while loop and display the sum to the user. If not, then we enter the while loop, and ask for another number.

This method is fine and all and guarantees that the user would have provided some input to our program. However, the biggest problem with it is that it requires code to be duplicated. If you look back at the snippet, you will realize that we must prompt the user for input twice, read the input twice, then do the addition twice. Once before the while loop, then again inside of the loop. Just mildly inefficient wouldn't you agree? But never fear, Do… While is here!

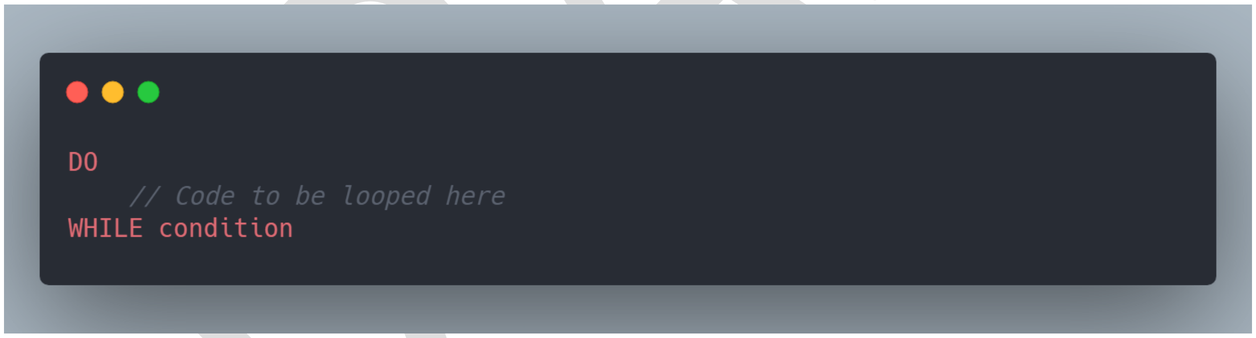
As mentioned, by using a Do while loop, we can guarantee that the loop will get executed at least once. This is because the loop is executed before the condition is checked. Look at the same problem above, but solved using a do while instead in the below snippet:



```
BEGIN
    DECLARE number and total as type Integer
    INITIALIZE total to 0

    DO
        DISPLAY "Enter another number. Enter 0 to exit"
        READ number
        total = total + number
    WHILE number != 0

    DISPLAY "The sum of the numbers you entered is ", total
END
```

Just like that, we remove our redundant lines, keeping only what we need. When the program begins, it will go inside of the DO, run all of the code inside there, and only then will it check the condition. If the condition is True, then it runs the loop again, however, if it is false, then the loop ends, and it displays the sum of the numbers. Neat isn't it.

The DO… WHILE loop generally follows this format:



```
DO
    // Code to be looped here
WHILE condition
```

Review

In this section, we have looked at 3 different types of loops, namely the For, While and Do While loops, along with how to create them and their use cases. With this, we can conclude this section on Repetitional Programming, and move onto the next section.

Functional Programming

Introduction

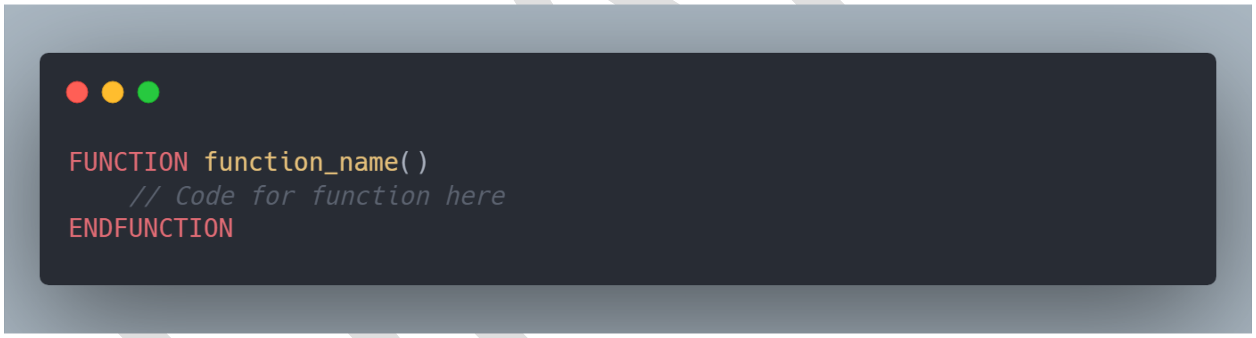
Functional programming is not one of the 3 basic programming constructs, but it is definitely a fundamental style that you **WILL** use throughout your computer science education. Functional programming makes use of a concept known as functions, which greatly improves the readability and manageability of your code, especially as it gets larger and more complex.

What are functions?

Functions are essentially sections of code that have been grouped together under a single name, which can then be invoked when it is needed and can return a value. Its main use is to remove redundancy and repetition from code, and uses a principle known as DRY (Don't Repeat Yourself), which allows us to run the same section of code without having to recode it each time. As we continue to explore functional programming, you will start to see why it is so helpful.

Creating a function

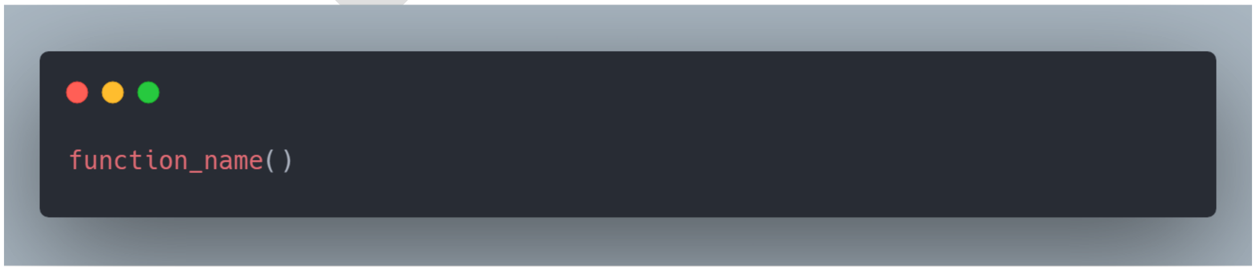
As mentioned, a function allows us to define a section of code that we can run at any point during our program, without having to type the same code repeatedly. To create a simple function in pseudocode, we can use the following format:



```
FUNCTION function_name()
    // Code for function here
ENDFUNCTION
```

This is known as the function definition. Much like how each entry in a dictionary has a name and an associated meaning, these lines allow us to give a name to our code (the meaning).

Now, whenever we want to run the code from within this function, known as calling the function, we can simply do the following:

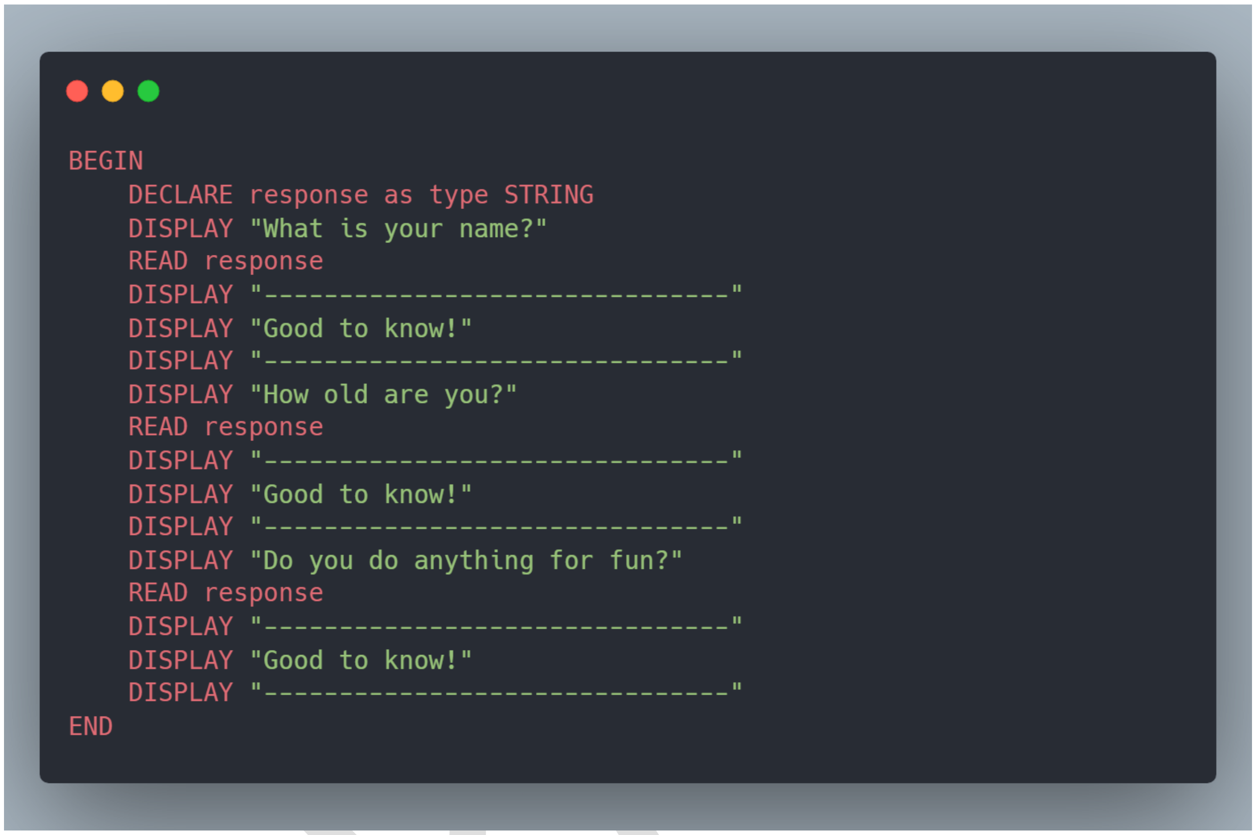


```
function_name()
```

Where *function_name* is the name of the function in both cases.

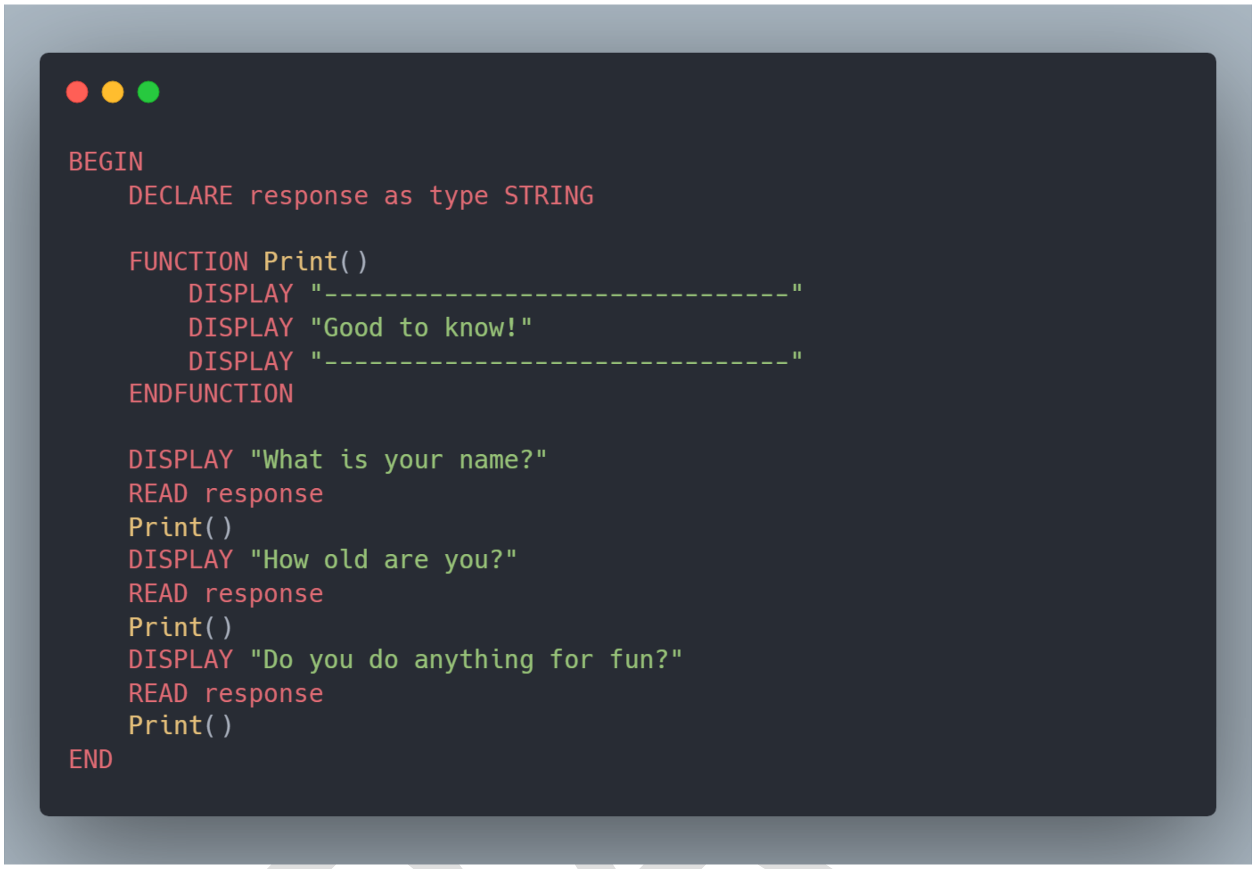
Example 1: Print Function

Say we wanted to display a certain set of messages multiple times throughout our program, we may do something like this:



```
BEGIN
    DECLARE response as type STRING
    DISPLAY "What is your name?"
    READ response
    DISPLAY "-------------------------------"
    DISPLAY "Good to know!"
    DISPLAY "-------------------------------"
    DISPLAY "How old are you?"
    READ response
    DISPLAY "-------------------------------"
    DISPLAY "Good to know!"
    DISPLAY "-------------------------------"
    DISPLAY "Do you do anything for fun?"
    READ response
    DISPLAY "-------------------------------"
    DISPLAY "Good to know!"
    DISPLAY "-------------------------------"
END
```

This is ok, I mean, it gets the job done. However, this is a lot of lines that we have to maintain now. What if we wanted to change the message from “Good to know!” to “Thank you for telling me”, for example. Then we would have to find every time we used that copy and pasted code and make the change there. What if we wanted to change the dashes to be another symbol? As you can imagine, managing these extra lines of code can turn out to be very costly and a waste of resources. By using functions, we can fix all those problems. Look at this snippet, which does the same thing as the previous one, but is so much neater.



```
BEGIN
    DECLARE response as type STRING

    FUNCTION Print()
        DISPLAY "-------------------------------"
        DISPLAY "Good to know!"
        DISPLAY "-------------------------------"
    ENDFUNCTION

    DISPLAY "What is your name?"
    READ response
    Print()
    DISPLAY "How old are you?"
    READ response
    Print()
    DISPLAY "Do you do anything for fun?"
    READ response
    Print()
END
```

The output from this code will be the exact same as the previous code. Wherever we call the Print function, by using **Print()**, the code within the function definition gets executed. In this case, the messages we set to be displayed are displayed to the user. Now if we wanted to change the message to say something else, all we must do is change the function definition, and the changes will be reflected across every place where we call the function. Neat and efficient.

Naming a function

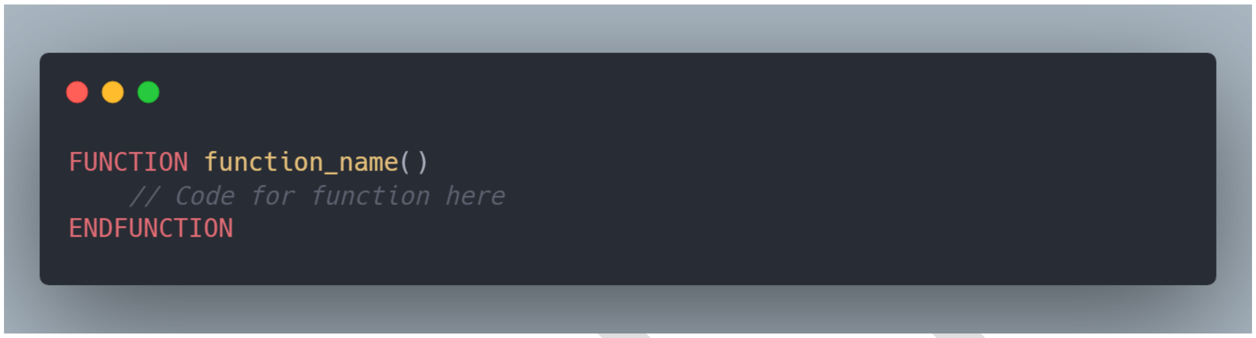
When naming a function, it should give a brief summary of what the function is meant to do. For example, if I were to name a function ***thing***, you would have no idea what it does unless you were able to look at the function definition. However, if I named my function ***sum***, even without looking at the definition you can make a good guess that the function will find the sum of give numbers. As with variable names, having a sensible name not only helps you, but any developer that has to work with your code in the future.

Even a name that may make sense to you when you do it, may leave you confused when you review your code to make improvements or fixes. I personally have been embarrassingly guilty of this, causing me to have to rewrite a large section of my code because I could not figure out what each part was supposed to do 😅.

Arguments and Parameters

So, you can use a function to call code that you stored in it from anywhere within your program to help improve maintainability and readability. Well, that's useful and all but, is that where the usefulness of functions end? From me saying that, I am certain that you already know the answer 😁. Where functions truly become powerful, is through the use of arguments and parameters.

Just a little while ago, we looked at the format needed to create a function. I.e.



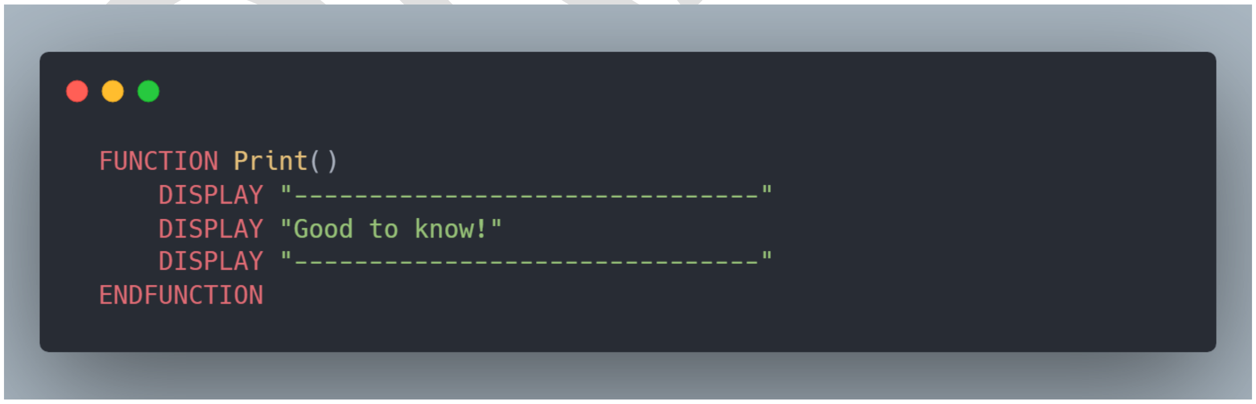
```
FUNCTION function_name( )
    // Code for function here
ENDFUNCTION
```

What may have gotten some of you confused or piqued your interests, was the parentheses, or brackets () as you may know them. You may have asked yourself what they were for and why you needed them. Well, now is the time to answer that question.

When we create a function, we can set up parameters. These are values that can change the behavior of the function. We do this by providing the name of the parameter inside of the parentheses, which then allows us to use the values within our function.

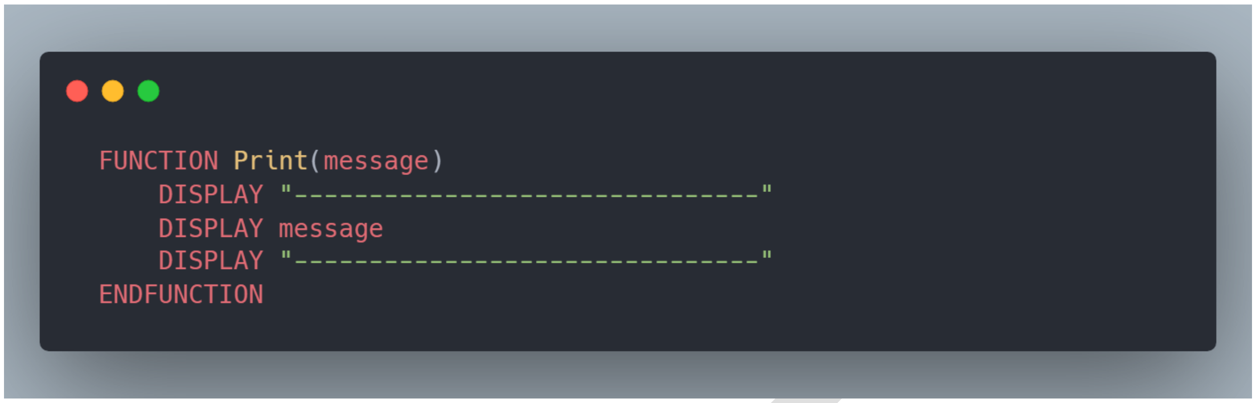
Example 2: Print Function

To showcase this, we can look once again at the Print function we made earlier.



```
FUNCTION Print( )
    DISPLAY "-------------------------------"
    DISPLAY "Good to know!"
    DISPLAY "-------------------------------"
ENDFUNCTION
```

What if instead of saying "Good to know!" every time, we wanted to say something different. Would it be better to just you sequential programming instead of a function? Would we have to make multiple functions with each change? The answer to both of those questions, is no. By providing a parameter to our Print function, we can modify the output however we please. For example:

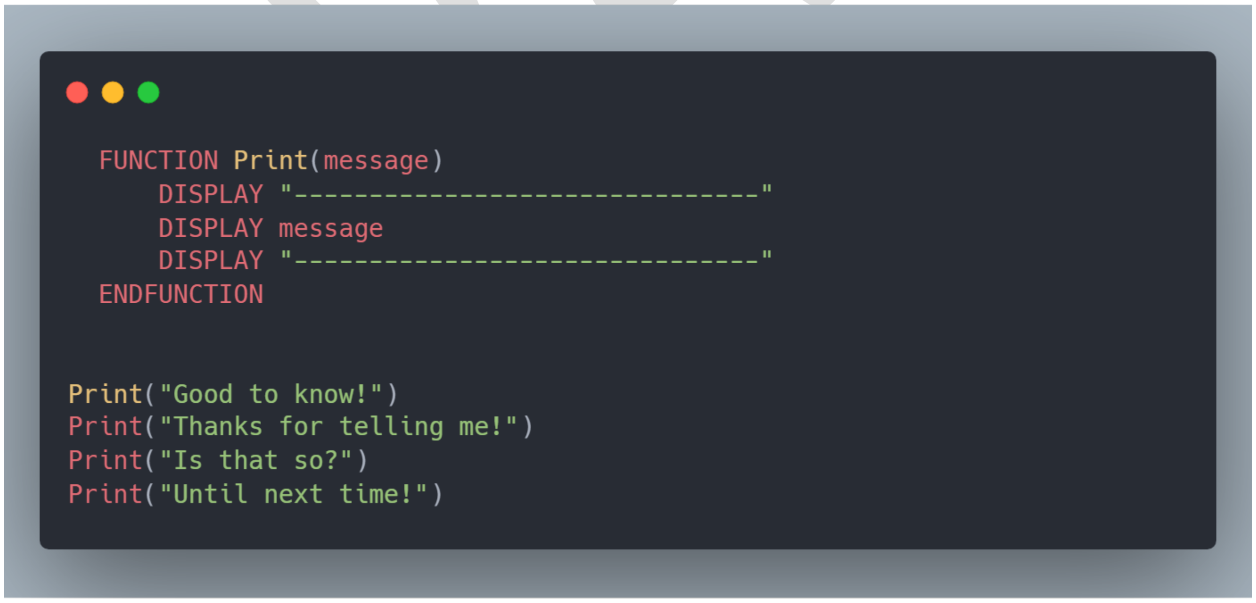


```
FUNCTION Print(message)
    DISPLAY "-------------------------------"
    DISPLAY message
    DISPLAY "-------------------------------"
ENDFUNCTION
```

Here, we created the parameter within the function definition, and called it message. Now within our function, we have access to this variable message and can now display it to the screen instead of the usual “Good to know!”

By now, you must be wondering, “But how do I get a value into the function for me to use it?”. The answer to this, are arguments.

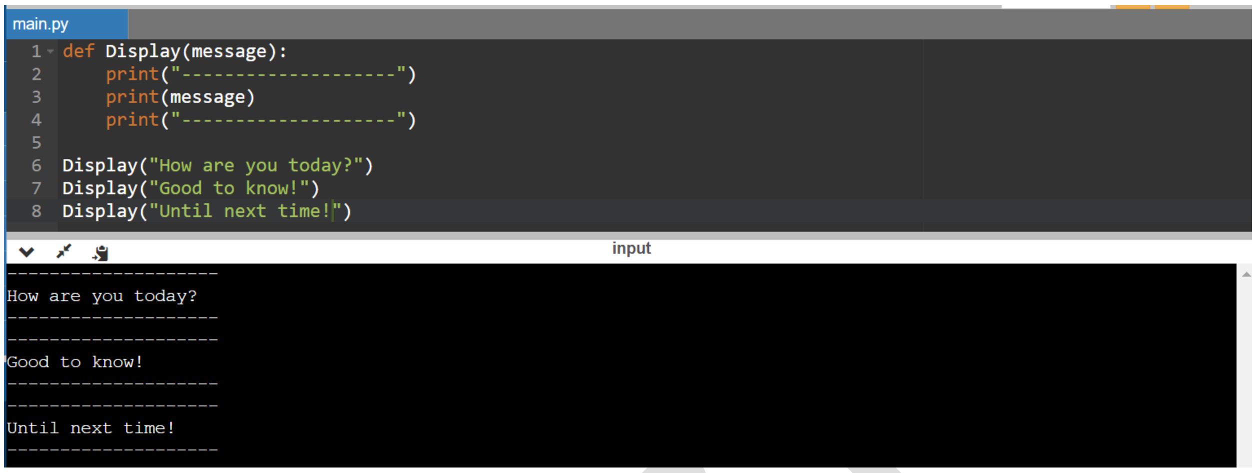
Recall, the parameters in our function definition, allow us to pass values from outside our function, inside of it, so we can use them as we please. These values that we pass in are known as the arguments. Similar to how the parameters are created in the function definition, arguments are passed in when we call the function. For example, if we wanted to display the following messages using the same format, the dashes, then the message, then the other dashes, we can do:



```
  FUNCTION Print(message)
      DISPLAY "-------------------------------"
      DISPLAY message
      DISPLAY "-------------------------------"
  ENDFUNCTION


Print("Good to know!")
Print("Thanks for telling me!")
Print("Is that so?")
Print("Until next time!")
```

Each time we pass a value into the brackets of the function call, we are passing that value to our function, which is then storing the value into the parameter called message, and then displaying it.

Here’s an example done in Python:



```python
def Display(message):
    print("--------------------")
    print(message)
    print("--------------------")

Display("How are you today?")
Display("Good to know!")
Display("Until next time!")
```

```
--------------------
How are you today?
--------------------
--------------------
Good to know!
--------------------
--------------------
Until next time!
--------------------
```

Figure 7: Using a function in Python

Recap

Parameters are set up in the function definition and allow you to pass values into the function to change how the function behaves.

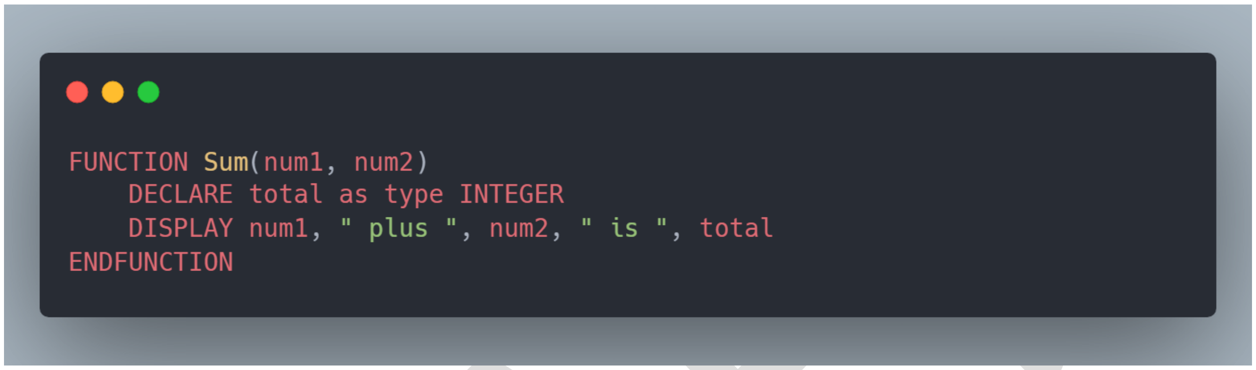
Arguments are values that you pass into the function when you call it to modify the function’s behavior.

Multiple Parameters and Arguments

In many cases, you may want to pass more than one argument into a function to further modify its behavior. For example, a function designed to find the sum of 2 numbers provided to it. To provide multiple parameters to one function, you separate each one with a comma. Same applies to passing in multiple arguments.

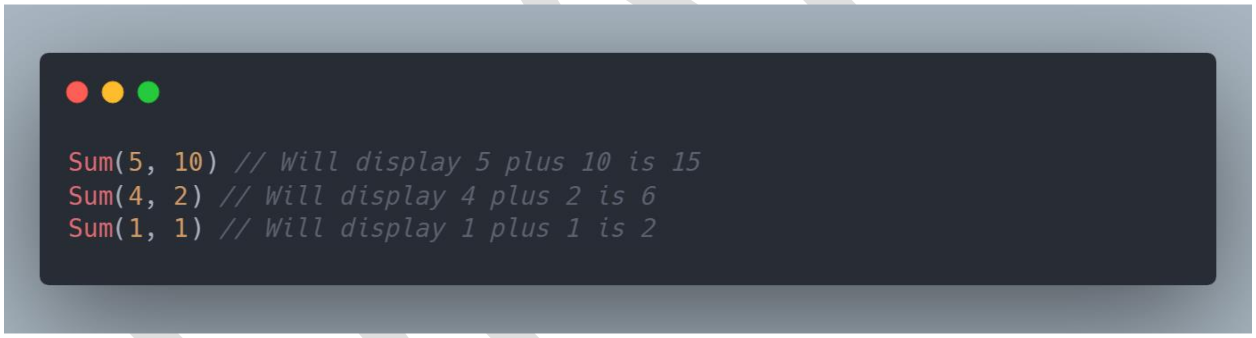
Take note that you may only pass in as many arguments as they are parameters.

Here's a look at a Sum function:



```
FUNCTION Sum(num1, num2)
    DECLARE total as type INTEGER
    DISPLAY num1, " plus ", num2, " is ", total
ENDFUNCTION
```

As was mentioned, this function takes in two numbers, and then displays their combined sum. If we were to call this function with numbers, we would do it like this:



```
Sum(5, 10) // Will display 5 plus 10 is 15
Sum(4, 2) // Will display 4 plus 2 is 6
Sum(1, 1) // Will display 1 plus 1 is 2
```

Note: You may have as many parameters as you need, as long as you provide just as many arguments.\

Review

In this section, you would have learnt about functions, how to create and name them and how to modify them using arguments and parameters. This covers all the fundamentals with regards to functions and concludes this section.

Review

In this section, we would have looked at the 4 different types of programming constructs, along with how and when to use each one. These constructs are arguably one of the most important parts of programming, as they will determine the best way for you to structure your code depending on the task that you have to complete.

End of Chapter 2

Congratulations on making it to the end of Chapter 2: Programming fundamentals. In this chapter, I have introduced you to all of the fundamentals that you need in order to really get off the ground and get started on your programming journey. Everything that we have covered here in this chapter is core and will be used in your day-to-day programming activities, so be sure to review it carefully and in detail.

The topics we have covered include:

- IPO (Input processing Output) and what it is
- Variables (Naming, creating, assigning, and using)
- Basic data types (Integers, Floats, Characters, Strings and Booleans)
- Operators (Arithmetic, Logical, Relational and Unary)
- Programming Constructs (Sequential, Repetitional, Selection and Functional)

Now that we have the fundamentals, we can step it up a bit by looking at Data Structures in the next chapter!