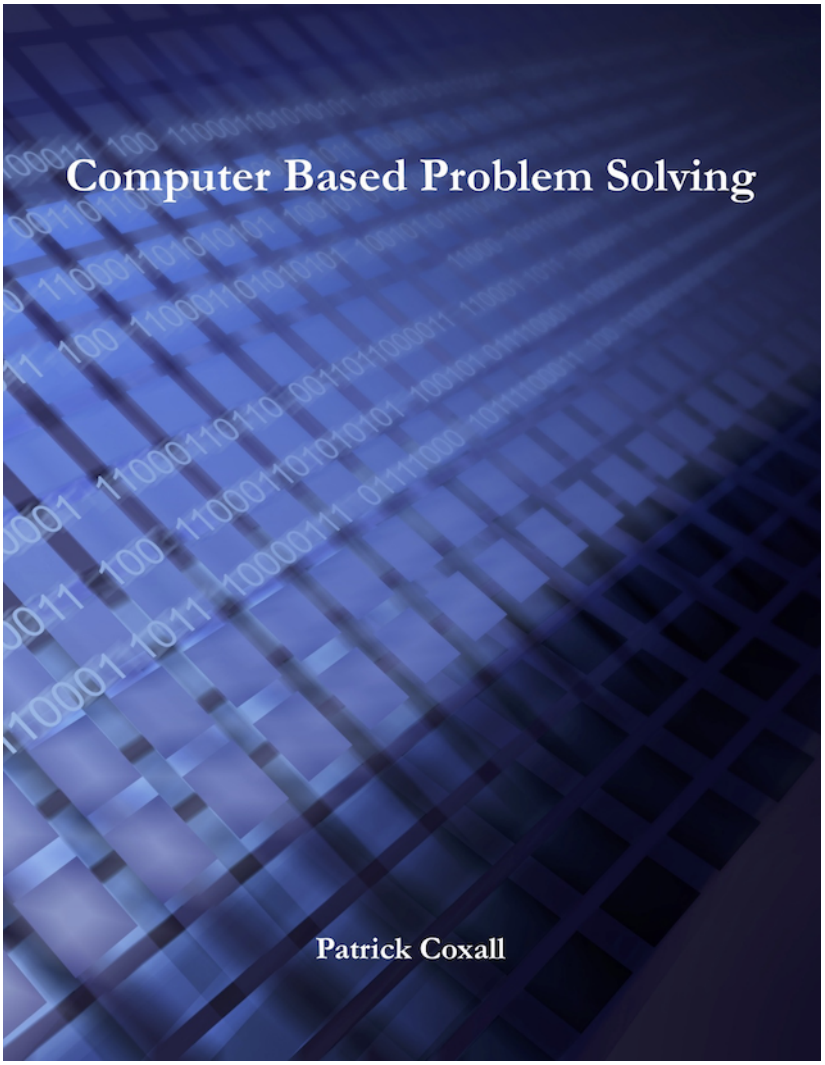
RTD$_{C}oxallDocumentation$

Release latest

Nov 16, 2020

CONTENTS



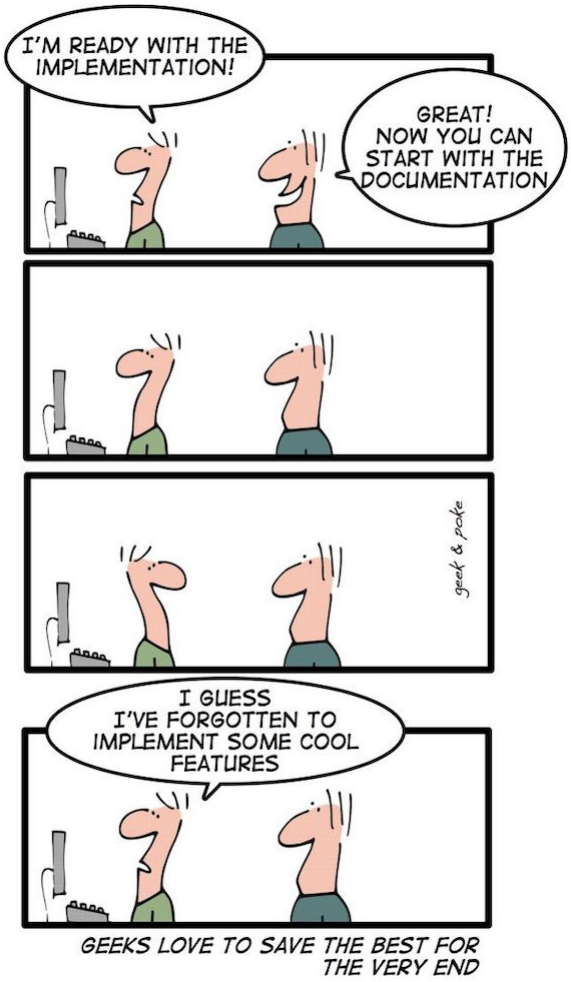
The goal of this book is to take students from the point of never having done any formal programming and lead them first through a structured method of problem solving (Input-Process-Output and Top-Down design), then into basic Structured Programming and then into the early basics of Object Oriented Programming (or OOP). If this book is used to teach a high school course in computer programming, there are likely many other learning outcomes that students are required to do that are not presented in this book. The focus of this book is strictly on solving problems with computer programming.

A PDF version of this textbook can be downloaded here, for offline reading.

CHAPTER

ONE

PREFACE



It should be remembered that the focus of this book is to teach students how to program, not to just teach them a programming language. To do this the focus is on "Problem Solving", using a computer program as a problem solving aid. Programming languages change over time and come and go but a good foundation of programming concepts and how to solve a problem will allow anyone to get over the syntax of a new programming language.

This book does not include any instructions on how to load, use, create IDEs or any other housekeeping of any particular language. There are many other resources that can aid both students and teachers alike for this.

Within the textbook you will see words or groups of words that are hyperlinked to Wikipedia. The point of linking to Wikipedia is to give additional information about a topic if the reader is unsure about the concept. Please note that I

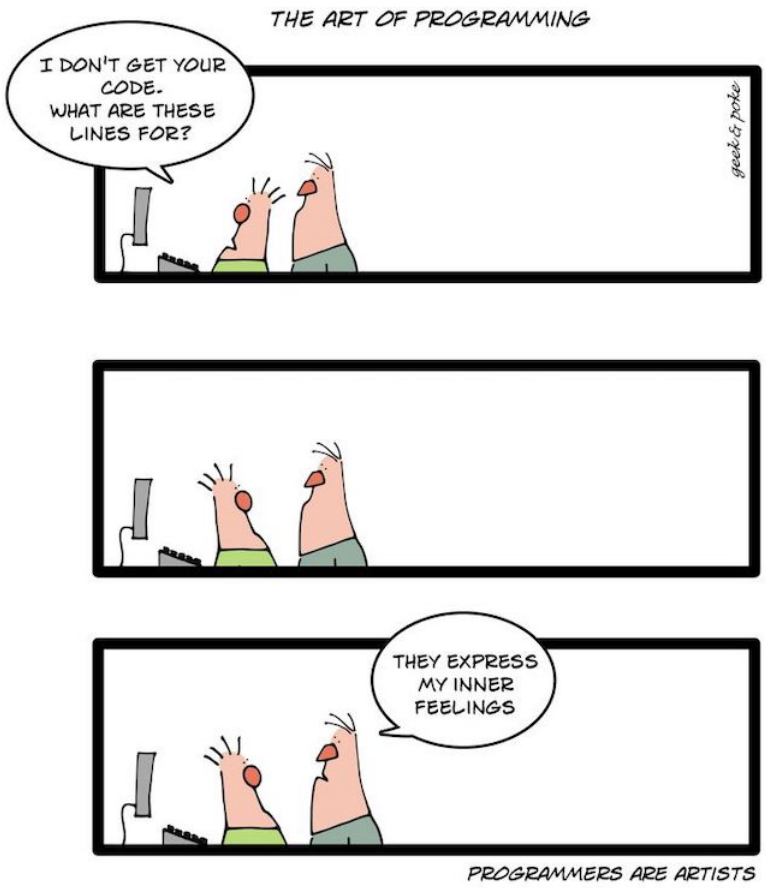
do not have control over what is placed on Wikipedia and although it seemed useful and correct when I looked at the link, these pages are changing all the time. Despite this, the information is usually correct and can be very helpful.

The cartoons are provided by Geek & Poke.

CHAPTER

TWO

INTRODUCTION



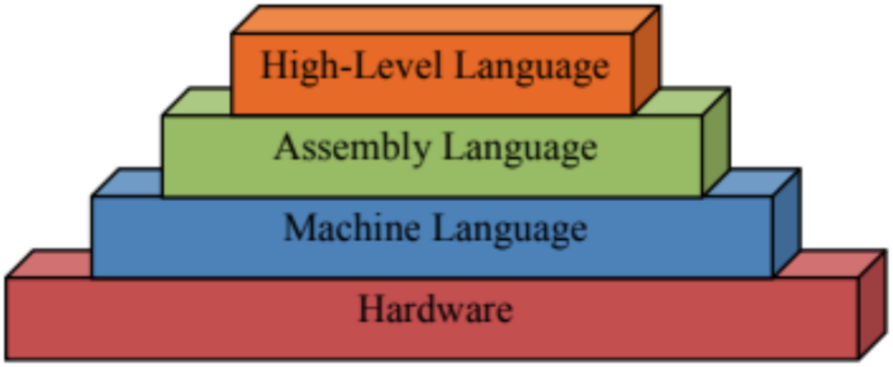
Problems have been around for as long as people have been around. The process of solving a problem is not something new. Using a computer to aid in solving a problem is new. Modern electronic computers have only been around since the Second World War (1939-1945), which might seem like a long time ago to you but in the history of the human

race it is a very short time. The purpose of this book is to help you learn to structure your problem solving method, so that you can consistently develop a verifiable solution that will solve a problem and in the process, use the computer to help you more easily and quickly solve that problem.

2.1 What is programming

Before you can actually start to write programs on a computer to help you solve problems, it would be nice to know what programming really is! We all use and see computers every day and hear people say how smart computers are. Actually, computers are not very smart at all! A computer, broken down into its most basic form is nothing more than a bunch of tiny electronic switches, called transistors, that can be set to either a 1 or 0 (on or off, also known as binary). By getting the computer to set these tiny switches on or off in a certain pattern, you can get the computer to actually do something useful, like show a picture on the screen that you have taken. The computer does not know how to do this by itself though.

To communicate with your friends, one way for them to understand what you mean is for you to talk to them (you could also use hand gestures but we have found out that it is not as reliable!). To keep things simple, you both usually talk in the same language. Since a computer is just a bunch of switches, it does not understand English, so you have to talk to it in a language that it does understand. Computers use a language called machine language, made up of just the 1's and 0's mentioned above. Trying to talk in machine language is quite difficult, easy to make mistakes in and tedious. People quickly realized they needed a better way to talk to computers. Assemble languages were created that used very simple instructions (like DCR C ; which decreases C counter by one). This assemble language is then run through a program that converts it to machine code. Eventually people wanted an even easier to understand language. To help people talk to a computer a high-level programming language is normally used, that is then translated into machine language so that the computer can understand what to do. This high-level language comes in many different variations and is normally just called a programming language and you have probably already heard of some of them (Java, C++, Python, ...). Just like there are many different languages that people speak around the world (English, French, Spanish, ...), there have been many different programming languages developed to help people instruct computers in what to do. The purpose of a programming language is to make it easier for a human to tell a computer what to do, by not having to talk machine language.



A person that uses a programming language to instruct a computer what to do is called a programmer. The programmer solves whatever problem they are working on, then writes the instructions that the computer is to follow in the programming language that they have chosen. Then the computer translates the instructions into machine language (the language that the computer actually understands) and the computer performs these actions.

2.2 Goal of this book

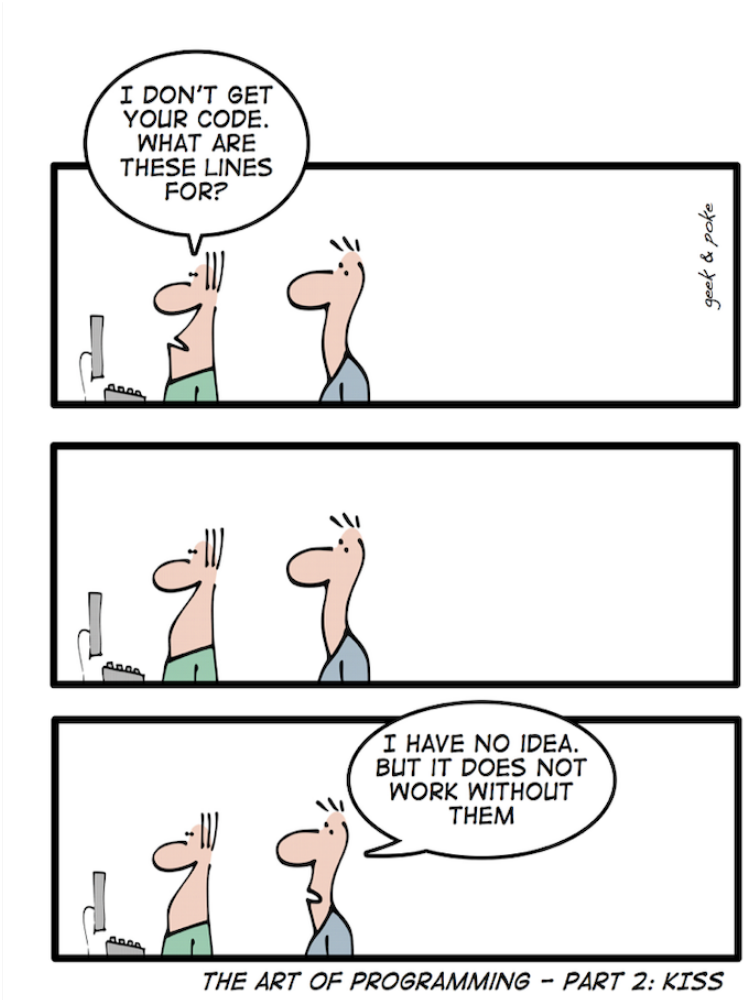
The goal of this book is to make you a "good" programmer. Despite the fact that a normal high school semester courses run for about 90 days, you will not become an expert programmer in just one semester. It has been said that it takes around 10,000 hours to really become proficient at anything and programming is no different. By the end of this book you will be very much on your way and have a good foundation in the skills you will need. The important thing to remember is that the point is not to teach you a specific programming language, since programming languages come and go and change over time. This is just like real languages.

Although it is the official language of the Catholic Church, not too many people go around on the streets and speak Latin to their friends. Many years ago, it might have been common but not today. The tools you will learn from this book are good programming techniques. These tools will be useful no matter what programming language you are using. Just like in the real world, you cannot be called a "linguist" if you only know one language. The same thing is true for a programmer; knowing more than one programming language is essential. The fortunate thing is that if you know how to solve problems for a computer and know one programming language, picking up a second one is much easier. The cornerstone of being a good programmer is to be able to solve problems in a logical and systematic way and hopefully have fun in the process.

CHAPTER

THREE

PROBLEM SOLVING



As previously mentioned problems have been around forever. The use of a computer to help in solving problems is new but computers do not solve problems, people still solve problems. Computers can be used to aid in solving problems but they are just a tool. People have been creating tools to help them solve problems for 1,000's of years. The key to remember is that a computer is just a tool, just like a hammer is a tool to help people put nails in a board.

My father is a joiner by trade but worked in construction, building houses when he first imigrated to Canada. He has

often told me stories of the "good old days" when they built houses completely with hand tools (using no electricity). They use to have competitions to see who could be the most accurate on estimating the length of a board, by cutting it first and then measuring it. They were usually within less than 1⁄2 inch (half the width of your thumb on a board 8 or 10 feet long!). It use to take dozens of men months to build a house this way. Then power tools were developed (electric drills, power saws, nailing guns, ...). Now a house can be built by a fraction of the men it use to take, in a fraction of the time. Power tools have revolutionized the housing industry.

Computers have also revolutionized many of the ways people solve problems, as compared to the past. The first modern electronic computer the ENIAC was built to calculate tables for firing artillery shells. (Some, and maybe correctly, argue that the first electronic computer was actually the Colossus). The ENIAC was developed because it took too long and there were too many mistakes when people were doing the mathematics to calculate the firing tables by hand. The ENICA could do the calculations in 30 seconds that it would have taken one person 20 hours to do! Today the same book of firing tables could be produced in a modern computer in a few seconds!

3.1 Steps in Problem Solving

There are many ways to solve a problem but having a process to follow can help make problem solving easier. If you do not think through a problem logically, then you end up just going around in circles and never solving it. The following is just one of many Six Step Problem Solving Systems, which can be used to solve any type of problem, not just ones that will be solved on a computer. The good thing is that the system translates nicely to computer problems, which is very useful, since the focus of the book is to solve problems on a computer.

The six steps in this system are:

1. What is the problem
2. Make a model
3. Analyze the model
4. Find the solution
5. Check the solution
6. Document the solution

3.1.1 What is the Problem

Before you can solve a problem correctly, you have to ensure that you understand the problem thoroughly. Many times you will have to go back to the source of the problem and confirm information or ask additional questions. You might have to have them restate the problem so that it is very clear what they are asking. Here are things to remember:

- What am I trying to find?
- What do I know / don't know?
- State the problem in your own words.
- Get them to restate the problem.

3.1.2 Make a Model

Making a model of a problem is a great way to see what is really going on and to lead you to a solution. It can show you patterns or you might recognize the problem from before. The model might be a drawing, picture, chart, a physical 3D scale model, or something else. Most "good" problems are too complex to be solved simply, they need to be broken down into smaller pieces, solve each of the smaller pieces and then bring all the small solutions back together to solve the original problem. Here are things to remember:

- Draw or create a model.
- Break the problem down into pieces.
- Is there a pattern?
- Have you seen something similar?

3.1.3 Analyze the Model

Once you have broken the problem down into more manageable pieces and made a model of the problem or the pieces of the problem, the next step is to understand what is really going on. If you do not become an expert at the problem, you might miss an important aspect. It is always a good idea to go back, not to the person that asked the question but the person that will be using the solution, to get information from them. Each piece might have a pattern that can be followed. You might have seen a solution for one of the pieces before. Here are things to remember:

- Ensure the model does what you think it does.
- Look for patterns you have seen before.
- Go back to the user to get more information.

3.1.4 Find the Solution

The hard part is now to find a solution. Hopefully you are well on your way by doing the above three steps. Can you find a pattern? Do you know the solution to one of the pieces? Can you find the solution somewhere (internet!)? In the world of programming there are book call, Patterns and Practices, that are full of common problems and their solutions. Once you have all your solutions, the next step is to bring them all together to a final overall solution to the original problem. Here are things to remember:

- Find a solution to each piece of the problem.
- Find other people's solutions to similar problems.
- Make sure all the pieces fit back together.

3.1.5 Check the Solution

You now (hopefully) have a solution to the original problem that you are pretty sure works. The next step is to ensure it actually solves exactly what the problem was. You might need to go through, step by step, to ensure it works. You might need to work through the solution and confirm the answer you get is correct. You might have to work through the solution several times and ensure you always get the same (or similar) answer. Here are things to remember:

- Is the answer reasonable?
- Work through the solution and check for errors.
- Go through the solution several times and compare results.

3.1.6 Document the Solution

So you have come up with a brilliant solution to a problem. If you do not share the solution with anyone, what was the point? You have to ensure that you answer is verifiable and reproducible, so anyone can use it. Here are things to remember:

- Document what the problem and solution is.
- Ensure anyone can follow your steps.

3.2 Example Problems

Here are some examples of problems and a possible solution using the six step method. Note that these problems are not necessarily problems that you would use a computer to help you solve. This is an important thing to remember, not all problems should be solved with a computer. Although this book is about solving problems using a computer, there are many problems that are better solved not using a computer and that is perfectly ok. Sometimes your job as a programmer might be to identify that you should not be solving the problem on a computer. Latter on these six steps will be translated into six step used to solve problems that a computer program will be used to help solve.

3.2.1 Folding Paper

How small can a piece of paper be made?

1. What is the problem?

The wording in this problem is a little vague and could be confusing. What is meant by "made"? Is it cutting, burning or shredding? You should go back to the source of the question to find out. You might find out that the real question is, "How many times can a piece of paper be folded?"

2. Make a model.

For this problem our model will just be our actual physical piece of paper. It is not always possible to make a model using that "real" item. What would happen if the question was to fold paper made of gold, can you afford gold leaf paper? You might have to use a suitable substitute.

3. Analyze the model.

You should check to make sure that your model will be correct. What is your size of paper? Would that not make a difference in how many times you can fold it? Maybe not? Once again you might have to go back to the source of the question and get more information. Maybe the real question is, "How many times can a piece of 8½"x 11" paper be folded?"

4. Find the solution.

In this case to find the solution we will take our piece of 8½"x 11" piece of paper and keep folding it until we cannot longer do it. Should we try more than once?

5. Check the solution.

It is always important to check your solution to see if it can be reproduced with accuracy. Maybe you could get someone else to fold a piece of paper and see how many times they can do it. Is it different from your answer? Maybe they have some special technique?

6. Document the solution.

Now that you have proven that a piece of 8½" x 11" can only be folded X number of times (where X is your answer), the next step is to document the solution so that other people can benefit from your analysis and can reproduce your experiment.

So what did you get as your answer? When you got someone else to do the experiment, did they get a higher number than you? Here are a few web links to also look at:

- Folding paper to the moon
- Can you really only fold a piece of paper 7 times

3.2.2 The Salmon Swimming

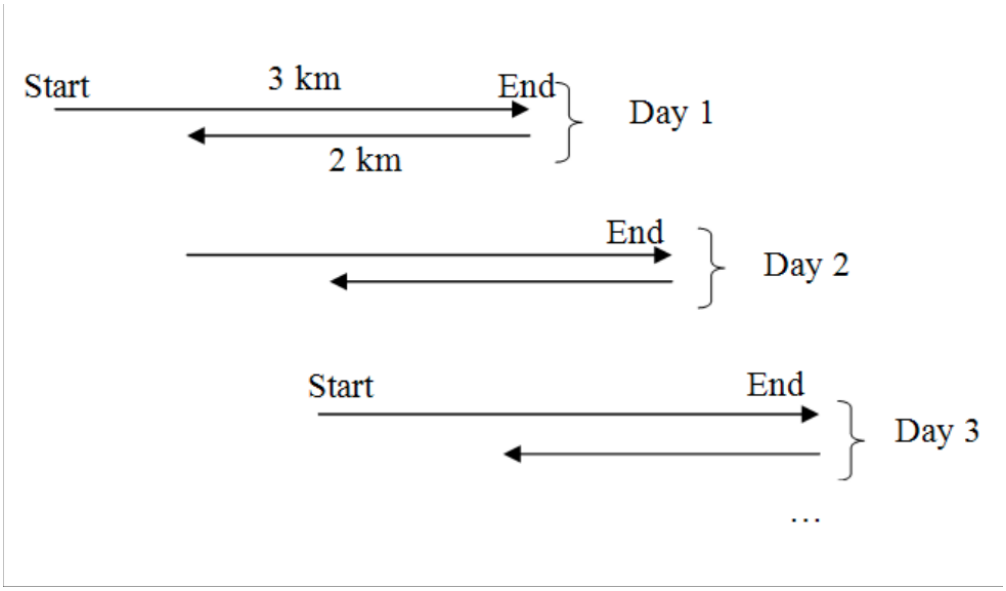
A salmon swims 3 km upstream and the current brings her back 2 km each day. How long does it take her to swim 100 km?

1) What is the problem?

The first thing that should be asked is, does the fish swim up stream 3 km during the day and fall back 2 km at night or does she swim continuously all day and would go 3 km if it was not for the 2 km current so she only ever gets 1 km? Once again you will have to go back to the source of the problem to find out. We will say she swims 3 km during the day and then drifts back 2 km at night.

2) Make a model.

For this problem our model will be a picture of a piece of what is happening.



3) Analyze the model.

You should check to make sure that your model will be correct. We will follow what is going on in a table:

Day Number	Distance at end of Swim	Distance after being moved back
#1	3	1
#2	4	2
#3	5	3
#4	6	4
#5	7	5
...	...	...
#96	98	96
#97	99	97
#98	100	98
#99	101	99
#100	102	100

4) Find the solution

In this case to find the solution we need to know how many days it took to get to 100 km. Your first reaction might be 100 days *BUT* if you look at the table on day 98 after the fish swam the 3 km, it is actually at 100 km mark, so that is the answer, 98 days not 100 days.

5) Check the solution

It is always important to check your solution. In this case since our solution came from the table, check to make sure there is no error in the table. It might be a good idea to let it sit for a few days and then come back to look at it or get somebody else to look at your solution and see if it is correct.

6) Document the solution

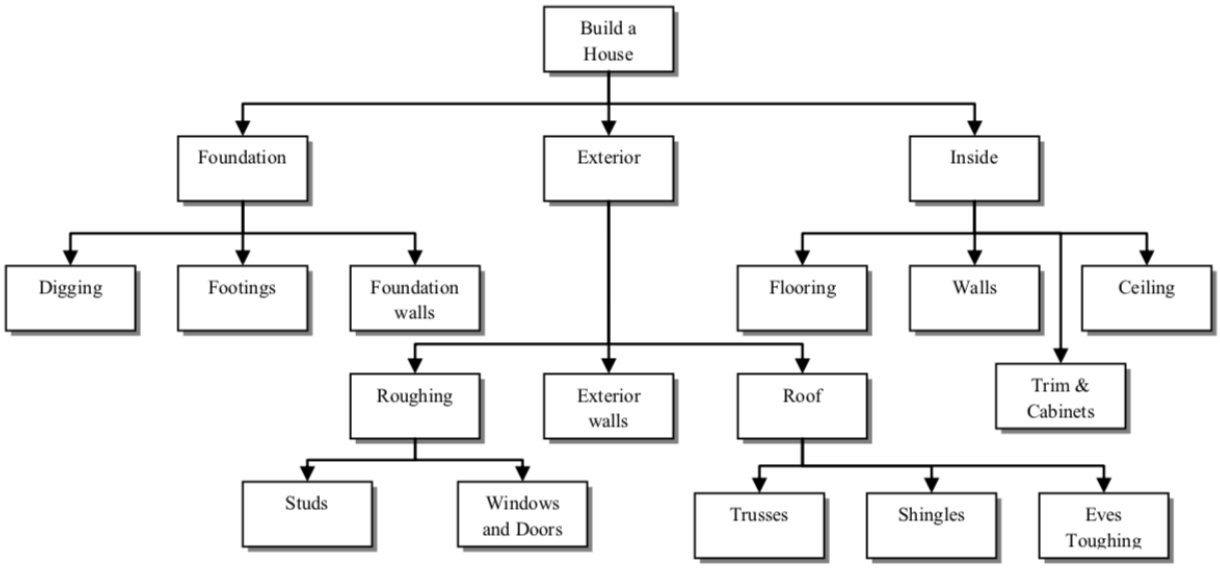
Now that you have proven that the answer is 98 days, make sure you document it, so that someone else does not have to figure it out but can just refer to your answer and check your solution.

Remember not to always go with your gut instinct and thing because it is following a patter you know the answer instantly without following through with the steps. Do all six steps and always check your answer.

3.3 Top Down Design

Top-down design is a method of breaking a problem down into smaller, less complex pieces from the initial overall problem. Most "good" problems are too complex to solve in just one step, so we divide the problem up into smaller manageable pieces, solve each one of them and then bring everything back together again. The process of making the steps more and more specific in top-down design is called stepwise refinement.

As mentioned before my father use to work in construction building houses. If someone gave you a piece of land and told you to, "Build me a house" you would not immediately go over and start nailing 2 x 4's together. Building a house is a very complex adventure, not to mention there are many rules, codes and laws that must be followed. To build a house you could break the project up into smaller jobs, plan and do each one of these jobs (in the correct order) and in the end you would have a house. You will notice in the following top-down design diagram that some jobs get broken down several times, until they are a manageable size.

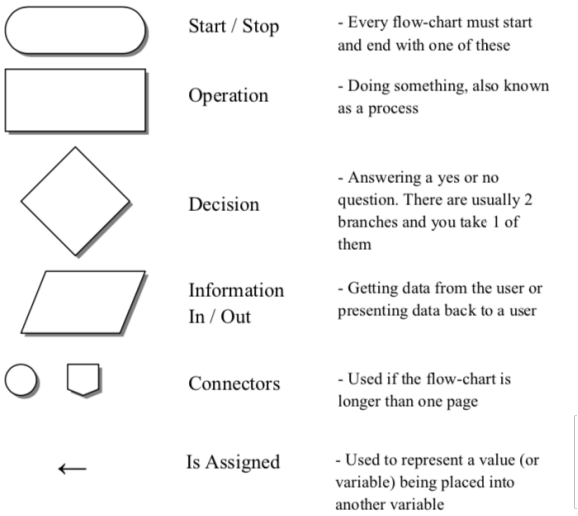


We will be using top-down design (and top-down design diagrams, like the one above) to help us understand a problem and all its components.

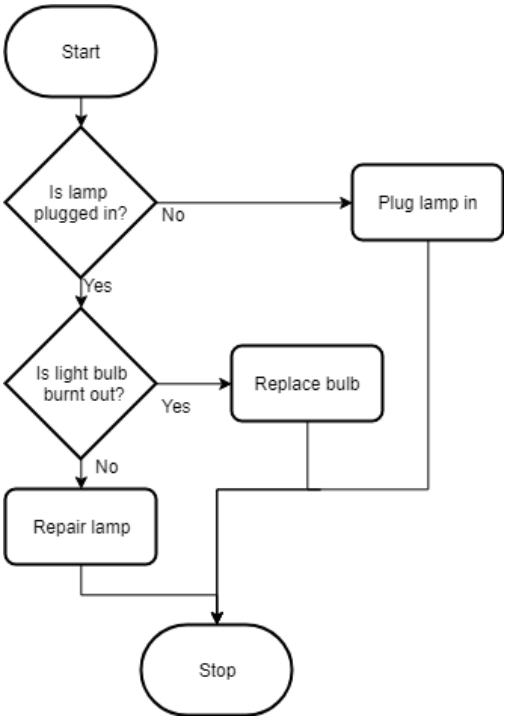
3.4 Flow-Charts

Some people think that there is no need to do flow-charts before writing a program; that you can just go to the computer and start writing code. Any "interesting" computer problem is so complex though, that without planning you would just end up spinning your wheels and have to throw out most of your code. In our six step problem solving model, the second step was to create a model and flow-charts are an excellent tool to make a model of what happens in most computer problems. Remember that a computer program is just a set of steps that the computer follows to solve a problem. A flow-chart is just a pictorial representation of a sequence of steps.

A flow-chart is a set of different shapes that each represent a certain type of action. These shapes are connected together with arrows so that you can see the flow of logic. The shapes are:



Here is an example of a flowchart for a none-computer based problem:



Flow-chart from Wikipedia

3.5 Pseudocode

Pseudocode is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudocode needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code.

We will be translating our flow-chart that we create from the previous step into pseudocode to aid us in writing our computer program. In general the vocabulary used in pseudocode should be the vocabulary of the problem, not written in "computer speak". A non-computer scientist should be able to read and understand what is going on. The pseudocode is a narrative for someone who knows the problem and is trying to learn how the solution is organized. Several keywords are often used to indicate common input, output, and processing operations.

Input: **READ, OBTAIN, GET**
Output: **PRINT, DISPLAY, SHOW**
Compute: **COMPUTE, CALCULATE, DETERMINE**
Initialize: **SET, INIT**
Add one: **INCREMENT, BUMP**

Here is the lamp example from the flow-chart section as pseudocode:

GET lamp does not work
IF (lamp plugged in == yes) THEN
    plug in lamp

ELSE
    IF (bulb burnt out == yes) THEN
        replace bulb
    ELSE
        repair lamp
    ENDIF
ENDIF

3.6 Computer Problem Solving

The initial goal of learning problem solving was to help us to solve problems that a computer program could be used for to help solve. The initial six step problem solving model that was presented can be used to help solve any type of problem. If we know that we are going to use a computer program to help solve the problem, the six steps can be translated into six steps that are more tailored for computer programming problems. They are the same basic six steps; they are just more focused on computer programming problems.

The table below shows the initial six steps that we have been using for generic problems. They are then translated into the six steps we will be following for computer programming problems.

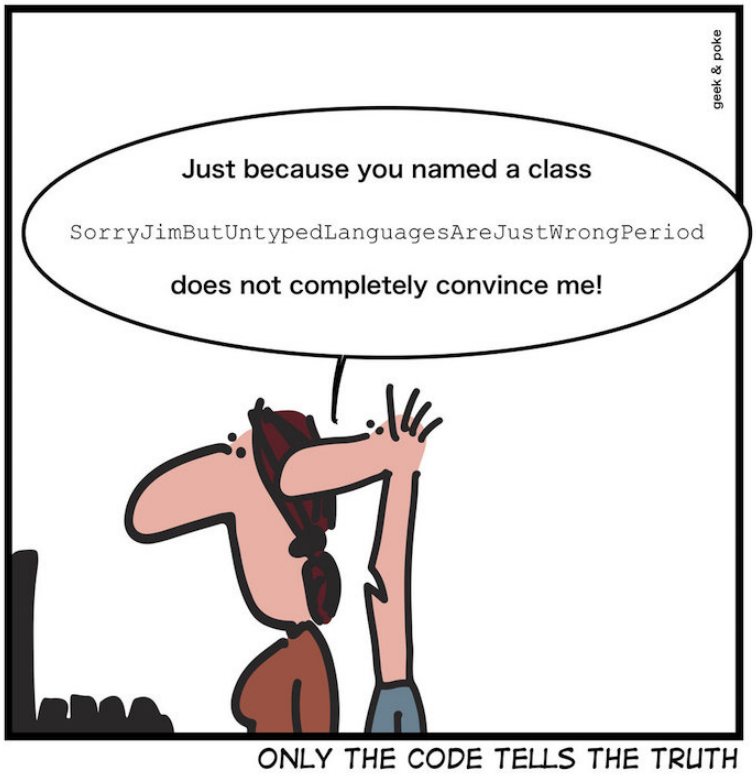
_	**Generic Step**	**Computer Programming Step**
#1	What is the problem	Top-Down Chart
#2	Make a model	Flow Chart & StoryBoard
#3	Analyze the model	Pseudocode
#4	Find the solution	Actually type in the code
#5	Check the solution	Style check & test for errors
#6	Document the solution	Document the code – comments & GitHub

These six steps are here to help you. Most people have the urge when they are given a programming assignment to just go to the computers and start coding. This is *NOT* a good idea. If you have not thought through the problem first and worked through these steps, you will make too many mistakes, get lost and waste too much time.

CHAPTER

FOUR

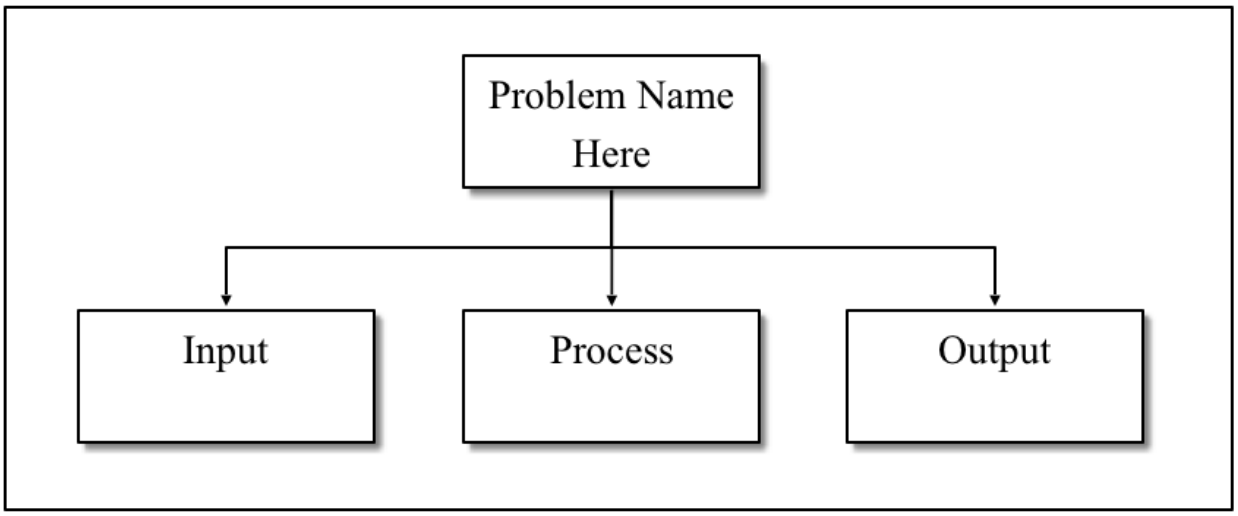
STRUCTURED PROBLEM SOLVING



Teachers often hear students complain that they "... don't know where to begin." when they are expected to solve what seem to be straightforward problems. Obviously they are not straightforward to the students for reasons that we are now beginning to understand, knowing where to begin is usually the hardest part. Structured problem solving is a set of tools to help you guide yourself though the process of solving computer related problems that seem to be impossible to solve.

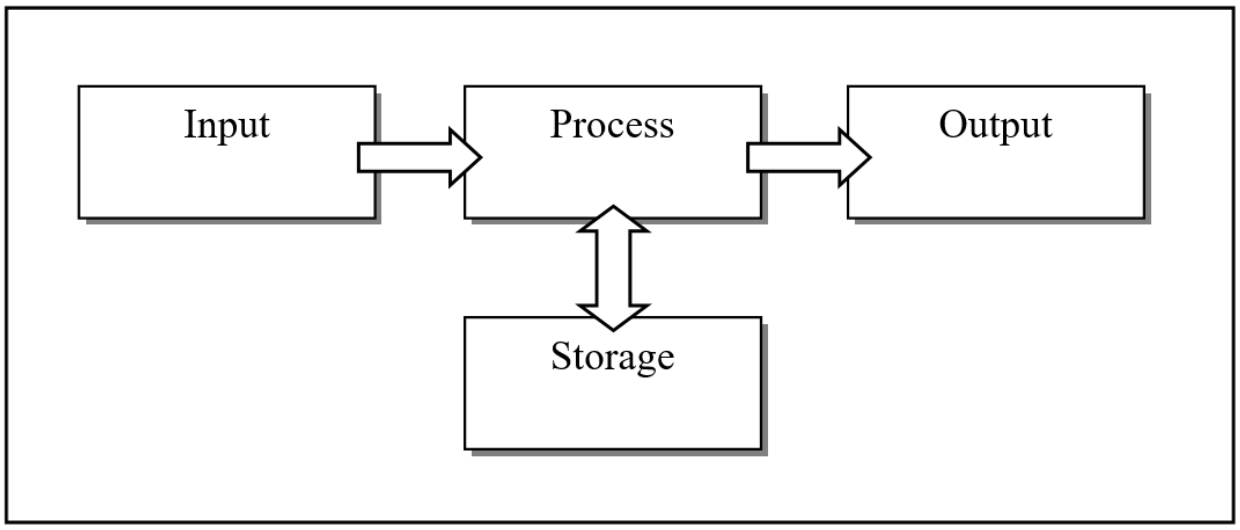
4.1 Top Down Design in Programming

As we have seen above, top-down design is a process of breaking a complicated problem down into simpler steps that actually can be solved. In programming one of the easiest models to follow for beginner programmers is the Input-Process-Output Model (IPO Model). You are most likely already familiar with this model, even if you do not realize you are using it. In life there are many examples of getting information, doing something with it and then returning the result. The most common example is probable the problems you do in math class every day. You get information, do calculations and then return the answer. We are going to use this model to help us solve our computer problems. The first step in our top-down design will always be to break the problem up into input-process-output.



4.1.1 Sometimes I need to Store Something

As mention above, "Input-Process-Output" is a very useful model to help solve problems. It is kind of misleading though because one very important point is missing, storage. What actually really happens is that we get information and then place it somewhere (write it down in a math problem or use a variable like x to hold some important number), we then process it (do some calculations), we get the answer and then give it back. We need storage as a temporary location to keep information. In really long problems many pieces of information might be kept, we might even have information that we just need to keep for some intermediate step. The real model looks more like:



In computer science programmers call these temporary storage locations varaibles.

4.2 Variables

A variable is a name that we use to represent a value that we want the computer to store in its memory for us. When solving problems, we often need to hold some valuable pieces of information we will use in our solution. From the "Input-Process-Output" model above, an example of variables we would be placing in storage is the input from the user. The pieces of information could be people's names, important numbers or a total in a purchase. We use a name that means something to us (and hopefully the people that come after us and read our source code, commonly referred to as just code) so that we know right away when we read our code what it is holding. In math class you might be familiar with equations that involve variables like "x or y". We would not name a variable x, if it is holding the number of students in class, we might call it numberOfStudents or number_of_students (depending on the style guide for a particular language). This has much more meaning to us and other people that also look at our code. Some people are still tempted to use a variable name like "x", becasue they say it will save space. But once our code is converted to machine language, it does not matter what you called your varaible it will be converted to something that takes the same space. So be a "nice" programmer and always use meaningful varaible names

Depending on the type of programming language you are using, you might need to declare your variable (warn the computer we will be using a variable before we use it) before you use it in a program. Some programming languages do not enforce this rule, other do. Since you are new to programming, it is really good programming style to always declare a variable before using it, if that is possible. The process of declaring a variable is called a declaration statement.

In most programming languages you will have an identifier, which is the name you are giving your variable (ex. numberOfStudents or number_of_students) and the data type. The identifier will be the way that you refer to this piece of information latter in your program. The data type determines what kind of data can be stored in the variable, like a name, a number or a date. In the computer world you will come across data types like integer, character, string & boolean. It is always important that you ensure you select the right kind of data type for the particular data that it is going to hold. You would not use an integer to hold your name and vice versa, you would not use a string to hold your age. The following is a table of some common built in types from several different languages that you might use:

Variable Type	Type Range
Boolean	True or False (1 or 0)
Unsigned Byte	0 to 255
Signed Byte	-128 to 127
Character	A single character (like A or % or @)
String	Variable length number of characters

Variable declaration usually should be grouped at the beginning of a section of code (sub, procedure, function, method...), after the initial comments. A blank line follows the declaration and separates the declaration from the rest of your code. This makes it easy to see where the declaration starts and ends. Ensuring that your code is easy to read and understand is as important in computer science as it is in English. It is important to remember that your code has two audiences, the computer that needs to compile or interprete it so that the computer can run your program and even more important, you and everyone else that looks at your source code that are trying to figure out how your program works. Here are some examples of declaring a variable:

4.3 Constants

There are times in a computer program where you have a value that you need the computer to save that does not change or changes very rarely. Some examples are the value of (3.14...) and the HST (the HST in Ontario is currently 13%, but it has changed once). When you have values like these, you do not want them to be saved as a variable (since they do not vary or change) but you place them in a constant.

Constants just like variables hold a particular value in memory for a programmer that will be used in the program. The difference is that the computer will not let the programmer change the value of a constant during the running of the program. This prevents errors from happening if the programmer accidently tries to change the value of a constant. It should always be declared, just as a variable is declared to warn the computer and people reading your code that it exists. Constants should be declared right before variables, so that they are prominent and easy to notice. Here are some examples of declaring constants:

4.4 Assignment Statement

Programs can have many variables. Usually information is gathered from the user, stored in variable, processed with other variables, saved back to one/some variable(s) and then returned to the user. Variables are changed or initially assigned a value by the use of an assignment statement. Assignment statement are usually written in reverse from what we are use to in math class. A variable on the left side of the assignment statement will receive the value that is on the right hand side of the assignment statement. Note that different programming languages use different symbols to represent the assignment statement (for example in Alpha it is" ←", in Pascal it is" :="). No matter what the symbol is, you always read it as, "is assigned". This is particularly important in languages like Visual Basic, where the assignment symbol is an equal sign (=) and people are use to reading this as "is equal to". C# also uses and equals sign (=) as its assignment operator. To make it even more confusing, the same symbol (=) is used in another context in Visual Basic and in that one it actually is an equal sign and is read as such. In C# the equals symbol is actually two equals signs next to each other (==). Here are a few examples of assignment statements:

4.5 Scope of Variables

Where a variable is declared is important because it determines its scope. The scope refers to where it is visible or can be used within a program. Usually you would declare a variable at the beginning of a function (for example a click event on a button or menu or the "main" function). Since it is declared at the beginning of a function, it can only be used within that funtion. Once the flow of your program exits this funtion, the variable is removed from memory (actually it is just de-allocate most likely) and can no longer be used. This type of variable is referred to as a local variable. Any other function in your program can not use or refer to this variable.

What if for some reason you needed a variable to be accessible to several different functions within a single program. In this case declaring it within a single function is no good. Another option is to declare the variable at the top of the form class or module, just before any function. If this is done then any function within that program can see and use this variable. This type of variable is called a global variable. Global variables should only be used when absolutely necessary; if only one function needs a variable, it should be declared within the function. This is good programming style and also saves computer memory. The following is an example where you can see variables with the same name, being used as global and local variables. Type it in and follow the variables by stepping through the program.

4.6 Sequence

Teachers often hear students complain that they "... don't know where to begin." when they are expected to solve what seem to be straightforward problems. Obviously they are not straightforward to the students for reasons that we are now beginning to understand, knowing where to begin is usually the hardest part. Structured problem solving is a set of tools to help you guide yourself though the process of solving computer related problems that seem to be impossible to solve.

4.6.1 Example Sequence Problem

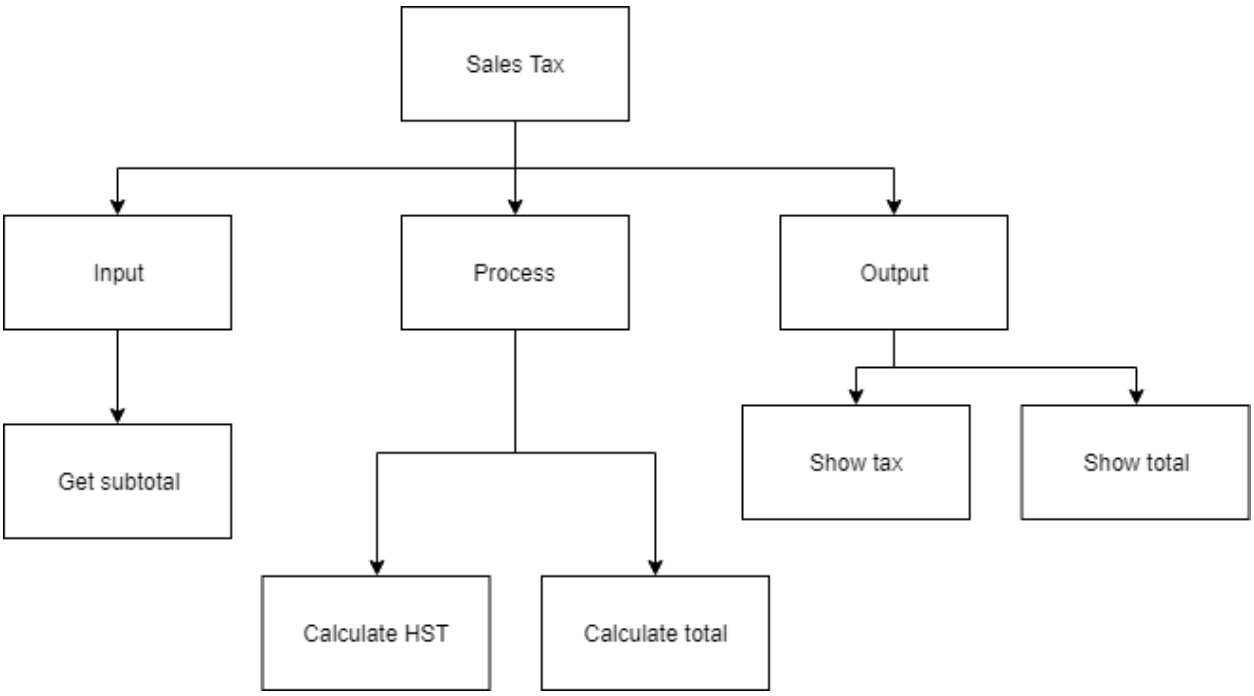
The following is an example problem that has been solved using the six step computer based problem solving method mentioned above. The goal is to show you how a sequential program works and also to show how the six steps are used to develop a program from a given problem.

Sequence Programming Example

Here is the problem: Write a program that will allow the user to enter a subtotal and the program will calculate the final total including tax.

1. **Top-Down Chart**

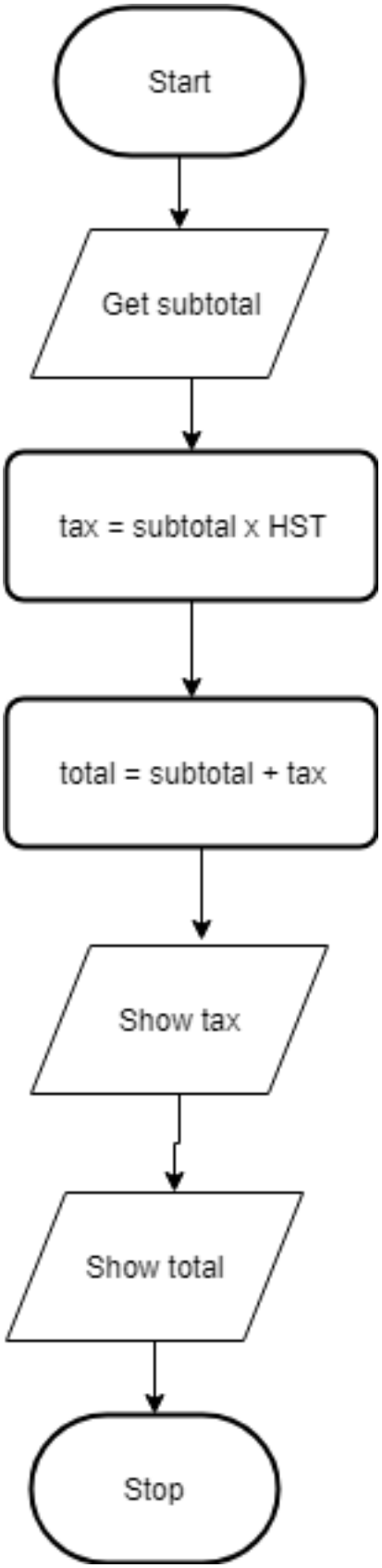
The first thing we need to find out is how to calculate tax in your particular province. If the program is to be used anywhere in Canada, it could get really confusing since each province has a different tax rate and some even calculate taxes differently than other provinces. To help simplify the problem, we are just going to do it for Ontario. The Harmonized Sale Tax (HST) in Ontario is currently 13%. The following is a top-down design breaking the problem up into smaller manageable pieces:



2. **Flow Chart**

The next step is to convert the top-down design into a flowchart. To help create the flow chart, use the bottom boxes of each of the arms, in order from left to right from the top-down design. This would mean Get subtotal, Calculate HST, Add 2 values together and Put final total. Remember that every flowchart has the "Start" oval at the top and the

"Stop" oval at the bottom. This is so people know where to begin and end. The arrows are used so people can follow the flow. The words in the flow charts are not full sentences but simplified pieces of code. Ensure that you include any formulas and use the variable names that you will plan on using in your code.


Start
Get subtotal
tax = subtotal x HST
total = subtotal + tax
Show tax
Show total
Stop

3. **Pseudo-code**

Pseudo-code converts your flowchart into something that more resembles the final code you will write. Once again though it is not code (hence the name pseudo-code), so it is generically written so that it can be translated into any language. It should be understood by anyone that can write a computer program, not just people that use the same programming language that you do. The first word on each line should be a verb (an action word), since you want the computer to do something for you. By convention the first verb is also in all caps (capital letters). Here is the pseudo-code for the problem:

GET subtotal from user
CALCULATE tax ← subTotal * HST
CALCULATE total ← subTotal + tax
SHOW taxes back to user
SHOW total back to user

4. **Code**

Once you have the pseudo-code done, the hardest part of solving the problem should be finished. Now you just convert your pseudo-code into the specific programming language you have chosen:

This program calculates total from subtotal and tax

5. **Debug**

It is hard to show the debugging step, since I ensured that the program above worked correctly before I pasted it into the page. When programmers write code it is extremely unlikely that it will work right away the first time. This is why the development environment has tools to help the programmer fix simple mistakes. The two main kinds of mistakes are syntax errors and logical errors.

In modern languages high level languages and IDEs, syntax errors are usually easy to see and fix. A syntax error is a piece of code that the compiler or interpreter does not understand. It would be like speaking to you and one of the sentences did not make any sense to you. A modern IDE will nicely place a squiggly line under the code (or some other way of showing you) it does not understand, so that you can fix the problem. A logical error is a lot harder to find. This is a problem with the way you solved the problem. The code will still compile or be interpreted and run but the program will give you the wrong answer (or maybe just the wrong answer some times!). There is not easy way to solve these problems than to step though your code one line at a time.

6. **Document the code**

This is hopefully not done just at the end of your programming but as you write your code. All the same it is good practice to go over you code at the end to ensure that someone else looking at it will understand what is going on. In the above example you can see that there is a comment at the start of the program and in the function as well. Also I have used a naming convention that is hopefully easy to understand what the variables are holding. In addition, the value of the HST is places in a constants, since they only change very infrequently.

The above six steps are an example of how you should go about solving a compute based problem. Ensure when you are given a problem, you do not make the mistake that most people do and go directly to the computer and start coding. If you have not first been able to break the problem down into smaller pieces and solve the problem on paper, going to the computer and starting to code will not help you. You will just end up going in circles, wasting time, creating bad code and getting nowhere. Programming is just problem solving on a computer but you have to have solved the problem before you actually get to the computer to help you get the answer.

4.7 Selection

If computer programs could only do just a linear sequence of steps and nothing else, they would not be very useful. One additional thing that computers are very good at doing is conditional control or making a decision, as long as you provide all the parts it needs to figure out the decision. This gives a computer program the ability to make a decision, based on a boolean expression.

4.7.1 Boolean Expressions

A boolean expression is an expression (or equation) that has two possible values or outcomes, either a "True" or a "False" (or you can look at it as a 1 or a 0). An example of a boolean expression is "A Volkswagen beetle is a car." Clearly this statement is true, so that is how it is evaluated to "True". You could also have "a frog is a mammal". This statement is not correct, so it evaluates to "False". You could also have mathematical expression, "3+2 = 6". This equation (and yes it is an equations so you read the "=" as "is equal") is not correct, so it is also evaluated to "False". There are many other types of expressions that can be checked, besides equality. You could have "3+2 <= 6". This time the inequality is evaluated as "True", since 5 is <= to 6. Some of the most common operators are:

Operator	Meaning
== or eq	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<> or !=	Not equal to

4.7.2 If...Then

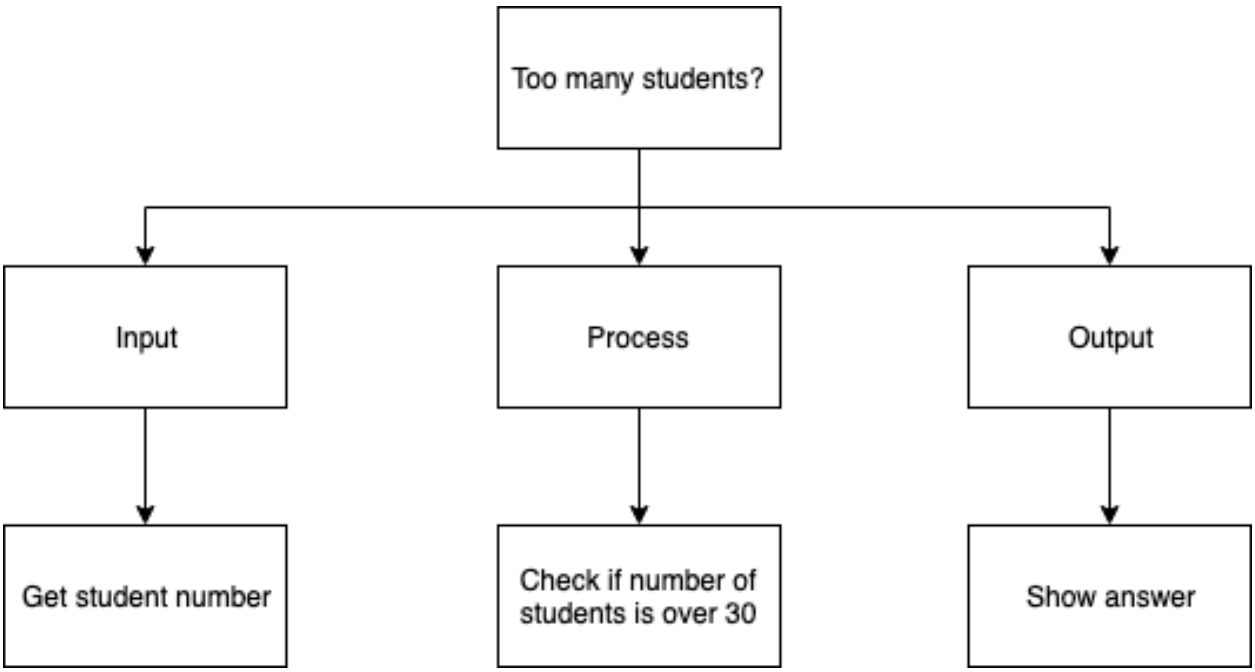
The If...Then structure is a conditional statement, or sometimes referred to as a decision structure. It is used to perform a section of code **if and only if** the condition is true. The condition is checked by using a Boolean statement. If the condition is not true (meaning false), then the section of code is not performed it is just passed over. The form of an If...Then statement is:

IF (boolean expression) **THEN**
    Statements to be performed
ENDIF

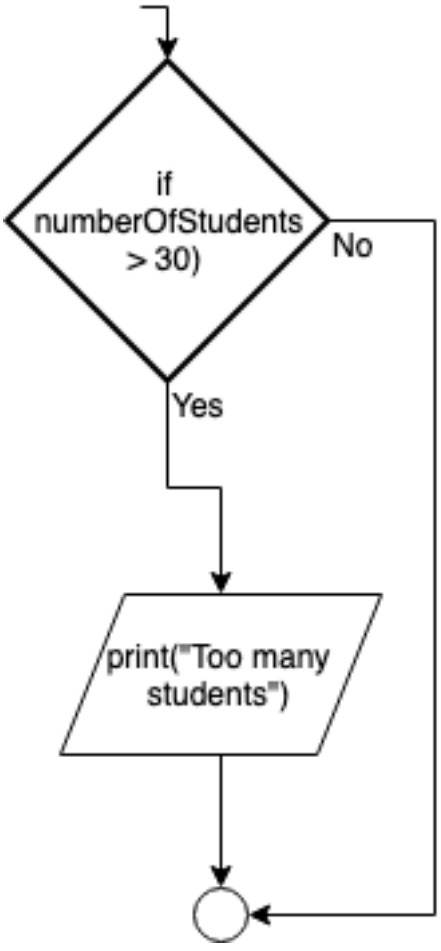
The indentation (usually 4 spaces, **NOT A TAB**) used in the If...Then statement is a coding convention used in almost every language. It is there to make the statement easier to read. It has no effect on how the code works and could be ignored; however, it is **REALLY BAD** programming style not to have it. You will also notice that some programming languages like to place the Boolean expression in brackets, while others do not. It is just style, but you should follow the language's style.

Here is a problem that can be solved using an If...Then statement. I have a class that can only hold 30 students because that is how many chairs I have. As the user to enter a number of students and tell me if I have too many students for chairs.

You top-down design will have a decision logic in it. You **do not** use a diamond in a top-down design, you still only use rectangles. Here is what a top-down design might look like for this problem:



Remember from the section on flowcharts, the diamond shape represented decisions. The If...Then statement is the translation of a decision in a flowchart to code. The above examples would look like the following in a flowchart:



You will also be using If... Then statements in pseudo-code. The above problem looks like this in pseudo-code. Note that you do indent when you are inside an If... Then statement in pseudo-code. Also note that "**IF**", "**THEN**" and "**ENDIF**" are all bold and caps:

GET number_of_students
IF (number_of_students > 30) **THEN**
    SHOW "Too many students!"
ENDIF

In the code examples below, if the variable numberOfStudents (or number_of_students) happens to be a number that is greater than 30 (say 32), the next line of code is performed (print("Too many Students!")). If the variable is not greater than 30 (say it is exactly 30), then the next line of code is skipped over and NOT performed.

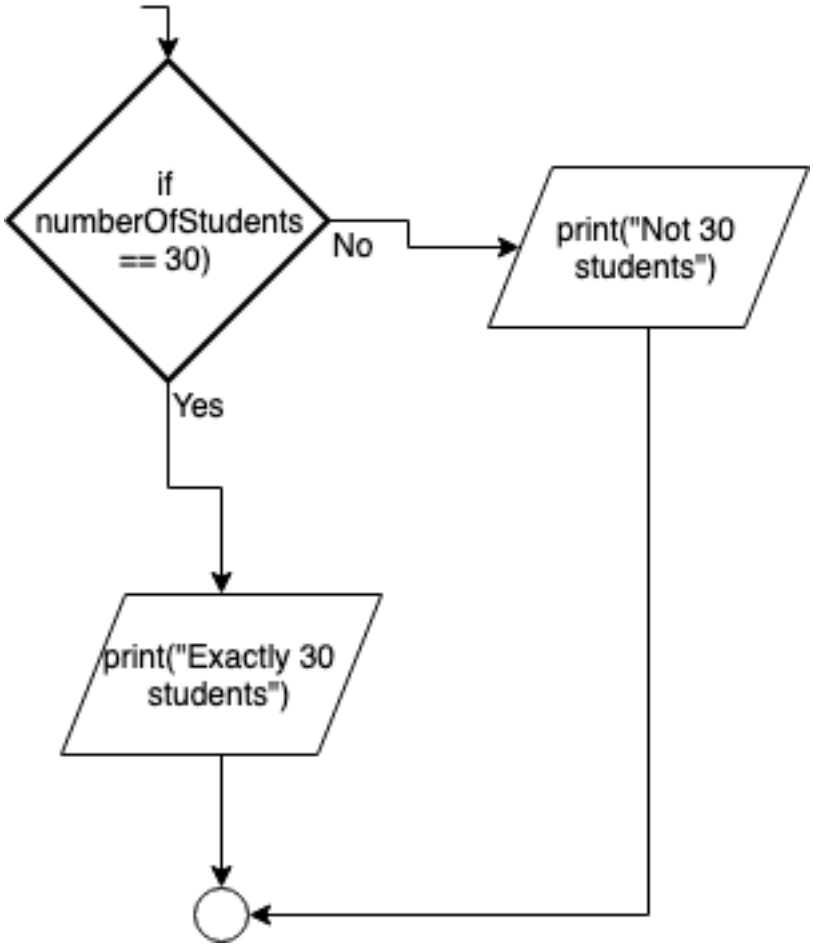
4.7.3 If... Then... Else

In the previous section we looked at the If... Then statement that is used for making a decision. When used a section of code is either performed or not performed, depending if the boolean statement is true or not. In some situations, if the statement is false and the section of code is not performed you would like an **alternative** piece of code to be performed instead. In this case an optional Else statement can be used. The If... Then... Else statement (in most computer programming languages) takes the generic form of:

IF (boolean expression) **THEN**
    Statements to be performed
ELSE
    Alternate statements to be performed
ENDIF

An example of what this would look like in a specific programming language is:

In the above examples, if the variable numberOfStudents happens to be exactly equal to 30, the next line of code is performed (print("Exactly 30 students!")). If the variable is not equal to 30 (say it is 32 or 17), then the next line of code is skipped over and **NOT** performed but the following line of code will be performed (print("Exactly 30 students!")). Once again the diamond shape represented decision, even if it has a statement if it is true and a different one if it is false. The above examples would look like the following in a flow-chart:



4.7.4 If... Then... ElseIf... Else

In some problems there are not just two different outcomes but more than two. If this is the case, then a simple If... Then... Else structure will not work. In these situations an If... Then... ElseIf... Else might be used. In this type of structure there can be more than just one Boolean condition and each is checked in sequence. Once a Boolean expression is met (the value is true), then the specified section of code for that Boolean expression is executed. Once executed, all other conditions are skipped over and the flow of logic goes right down to the bottom of the structure. It is important to remember that **one and only one** section of code can be executed. Even if several of the Boolean conditions happen to be met, the first and only the first one will be evaluated and run.

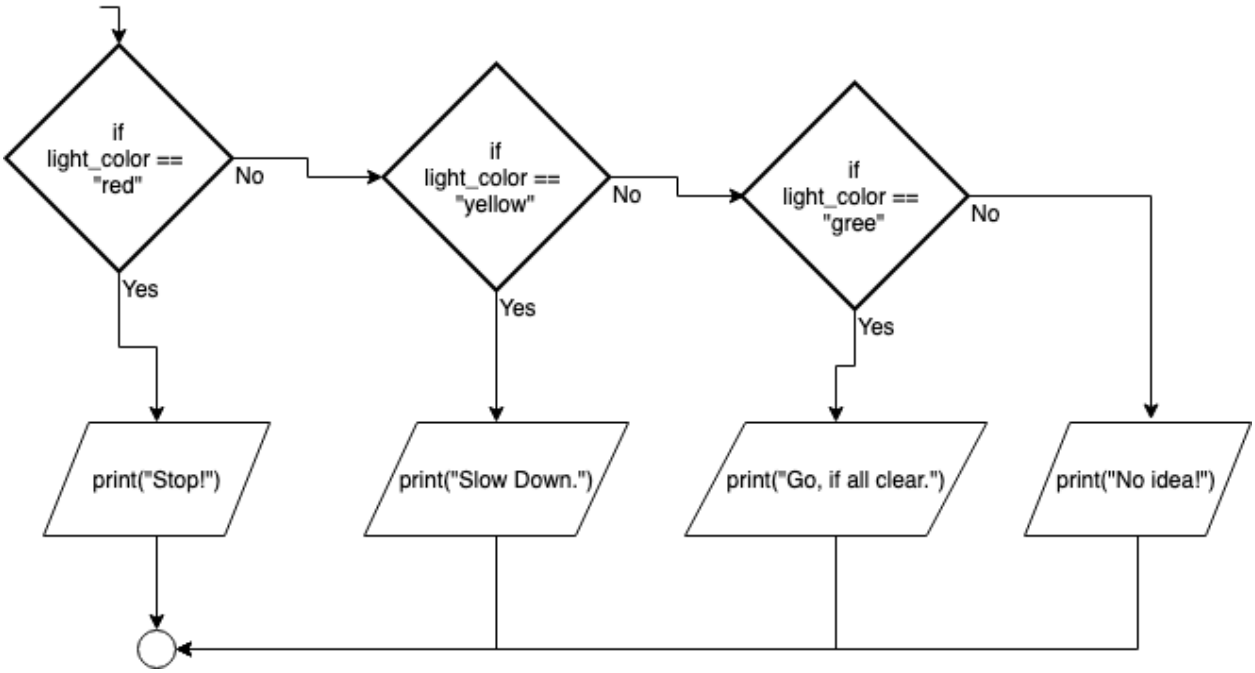
The structure can contain many ElseIfs, as many as the programmer needs. Another optional piece of the structure is the "Else". If none of the above boolean conditions are met, the programmer might want a section of code to be executed. If this is the case, then the code is placed in the else section. Any code in the else section is run if and only if none of the Boolean expressions were met. It should also be noted that there is no Boolean condition associated with the else. That is because it is run only if all the above boolean conditions are not met. The If... Then... ElseIf... Else statement (in most computer programming languages) takes the generic form of:

IF (boolean expression #1) **THEN**
    First potential statement to be performed
ELSEIF (boolean expression #2) **THEN**
    Second potential statement to be performed

ELSEIF (boolean expression #3) **THEN**
    Third potential statement to be performed
...
ELSEIF (boolean expression #n) **THEN**
    Nth potential statement to be performed
ELSE
    Alternate statements to be performed
ENDIF

An example of what this would look like in a specific programming language is:

In the above examples, if the variable colourOfLight is red, yellow or green than the appropriate section of code is executed. If the variable does not equal any of these, then the last statement is executed, "No idea!" The above examples would look like the following in a flow-chart:



4.7.5 Select Case

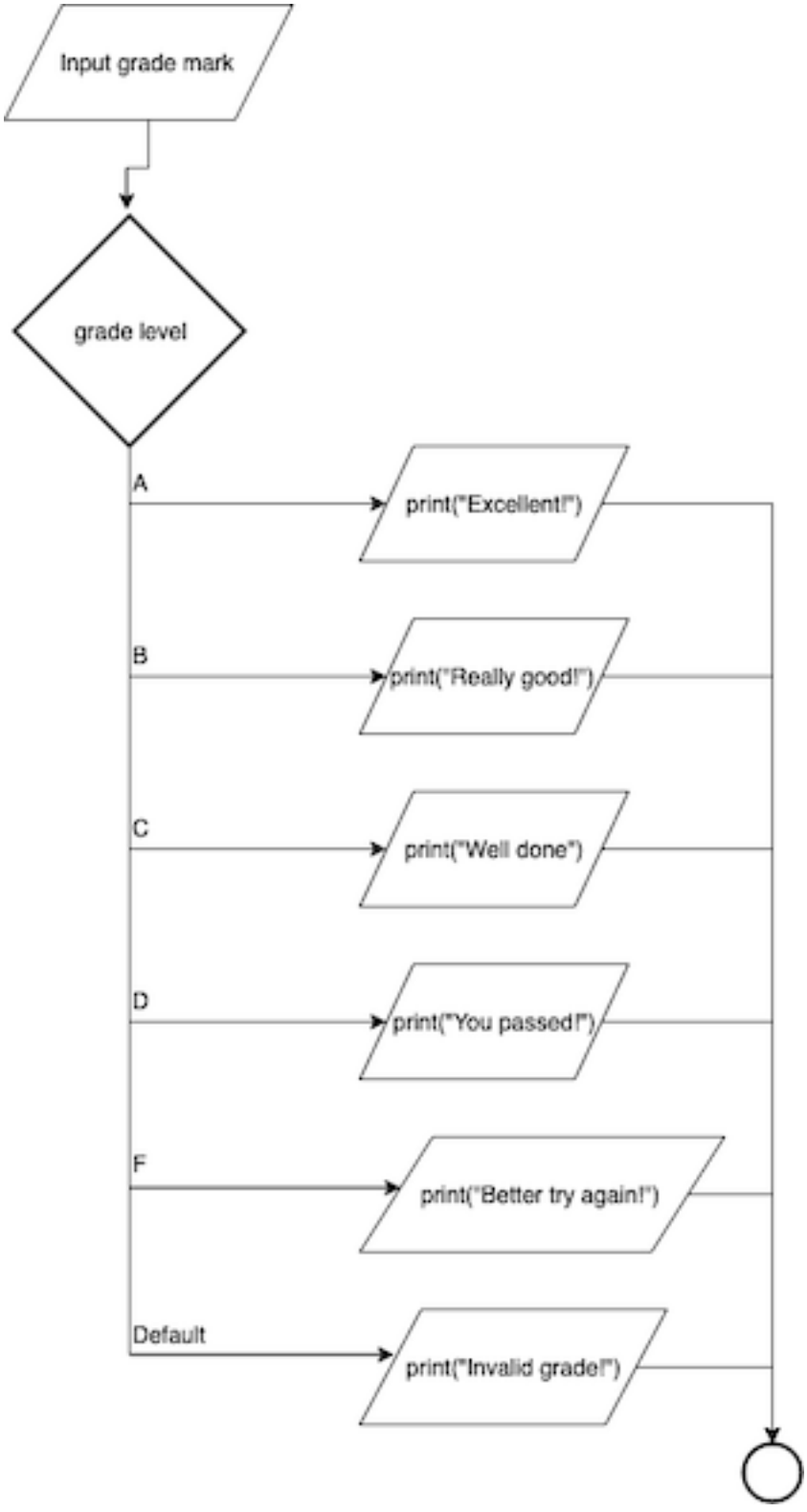
As you have seen from the If...Elseif...Elseif...Else statement, when there are many choices, the structure can be hard to follow. Some programming languages have an alternative structure when this happens. The Select Case or Switch Case statement is also a decision structure that is sometimes preferred because code might be easier to read and understand, by people.

The Select Case structure takes a variable and then compares it to a list of expressions. The first expressions that is evaluated as "True" is executed and the remaining of the select case structure is skipped over, just like an If...ElseIf... statement. There are several different ways to create your expression. You can just use a value (a single digit for example), several digits, a range or having a regular expression (Is < 10). Just like the If structure, there is an optional "Else" that can be placed at the end as a catch all. If none of the expressions is evaluated to "True", then the flow will go to the else. The general form of a Select...Case statement (in most computer programming languages) takes the generic form of:

SELECT (variable)
    CASE valueOne
        //statements
    CASE valueTwo
        //statements
    CASE valueThree
        //statements
    ...
    ELSE //optional
        //statements

An example of what this would look like in a specific programming language is:

In the above examples, if the variable gradeLevel is "A" it will print out "Excellent!", and so on for each letter of grades. If the variable does not equal any of these, then the last statement is executed, Invalid grade". The above examples would look like the following in a flow-chart:


Input grade mark
grade level
A
print("Excellent!")
B
print("Really good!")
C
print("Well done")
D
print("You passed!")
F
print("Better try again!")
Default
print("Invalid grade!")

4.7.6 Try Catch

When a runtime error or exception occurs, the program will terminate and generate an error message. This is not exactly ideal for the user. What would be better is if we could catch these errors and then do what is necessary to get fix the problem. A common example is when we ask the user to enter a number, but for some reason they entered text. Ideally we would not want the program just to crash, we would want to explain to the user they entered something incorrectly.

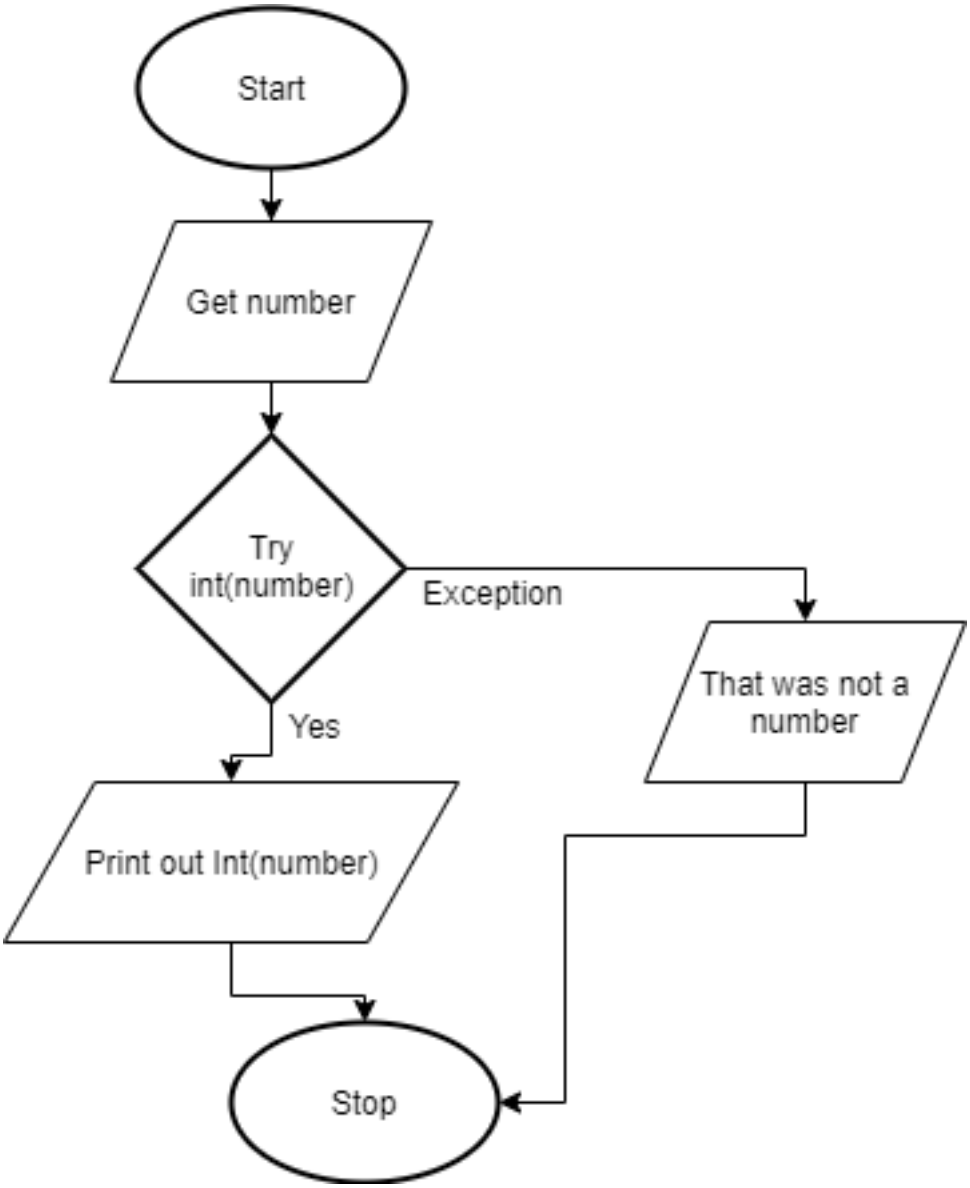
To catch these runtime errors, a portion of code is enclosed in a try-block. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored. The try statement (in python) takes the generic form of:

TRY
    some statement(s) to be performed
EXCEPT (type of error)
    some statement(s) to be performed
ELSE
    some statement(s) to be performed
FINALLY
    some statement(s) to be performed
END

You can define as many exception blocks as you want (e.g. if you want to execute a special block of code for a special kind of error). You can use the else keyword to define a block of code to be executed if no errors were raised. The finally block, if specified, will be executed regardless if the try block raises an error or not.

An example of what this would look like in a specific programming language is:

In the above examples, if you do enter in an integer, it will let you know. If you enter in a string for example, the program will not crash, but give you a warning. The above examples would look like the following in a flow-chart:



4.7.7 Compound Boolean Expressions

Just before we looked at the If ... Then statement we looked at Boolean expressions. Boolean expressions have two and only two potential answers, either they are true or false. So far we have looked at just simple boolean expression, with just one comparison. A boolean expression can actually have more than one comparison and be quiet complex. A compound boolean expression is generated by combining more than one simple boolean expression together with a logical operator And or Or. And is used to form an expression that evaluates to True only when both operands are true. Or is used to form an expression that evaluates to true when either operand is true. Here is a truth table for each operator:

AND Truth Table

A	B	A AND B
True	True	True
True	False	False
False	True	False
False	False	False

OR Truth Table

A	B	A OR B
True	True	True
True	False	True
False	True	True
False	False	False

In some programming languages the operators are simpley the words "AND and OR". In others they are "&&" for AND and "||" for OR. The following are some examples of compound boolean expressions:

Besides these two logical operators, there is one more, the NOT. NOT is used most often at the beginning of a Boolean expression to invert its evaluation. It does not compare 2 values but just inversts a single one.

NOT Truth Table

A	NOT(A)
True	False
False	True

For example:

4.7.8 Nested If Statements

Sometimes a single if statement, even a long If...Then...ElseIf...ElseIf...Else is not a suitable structure to model your problem. Sometimes after one decision is made, there is another second decision that must follow. In these cases, if statements can be nested within if statements (or other structures as we will see later). Here is a problem:

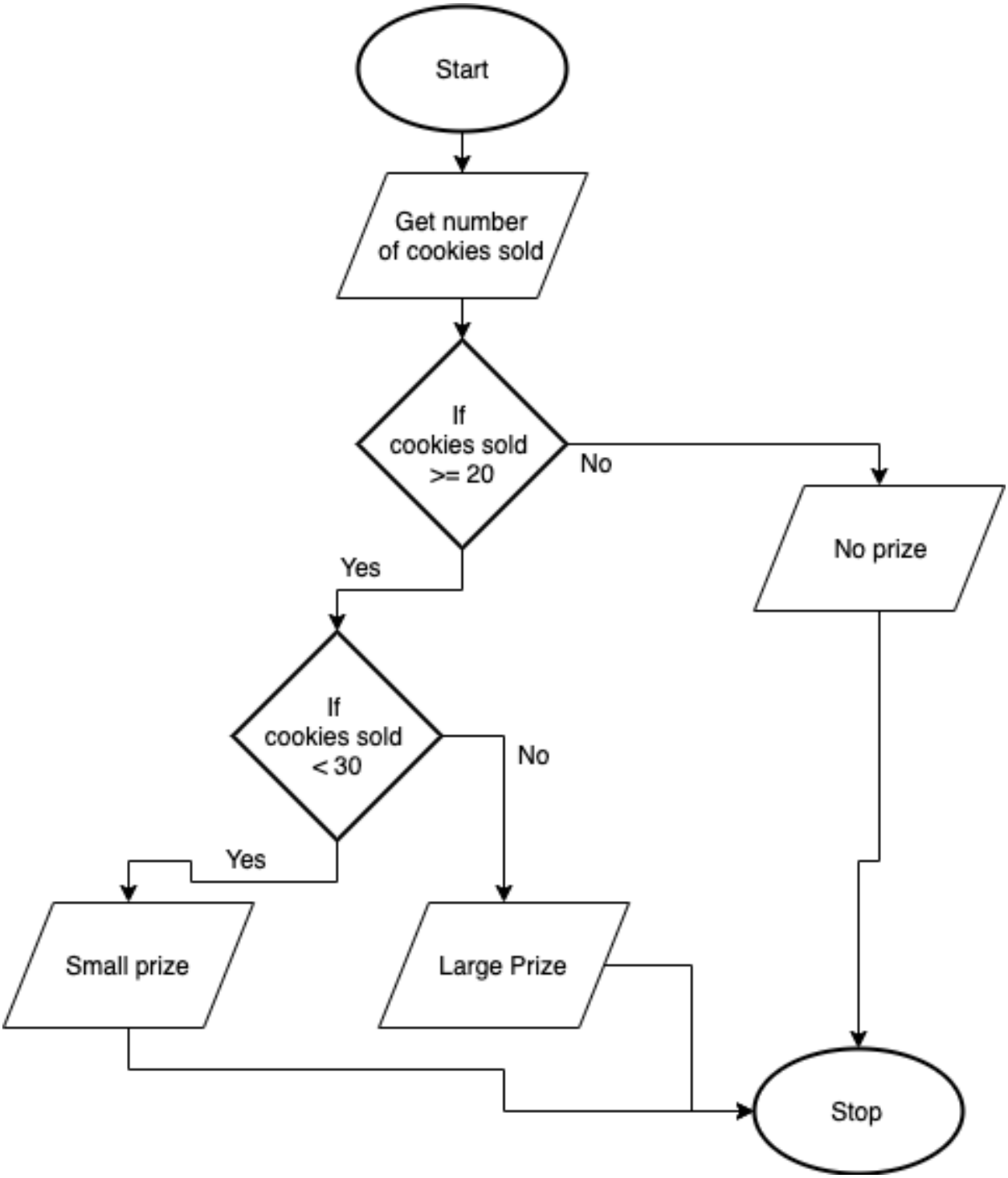
The nested if statements (in most computer programming languages) takes the generic form of:

IF expression1:
    statement(s)
    IF expressionA:
        statement(s)
    ELSE
        Alternate statements to be performed
ELSE
    Alternate statements to be performed
ENDIF

A school is going to sell chocolate bars to raise money. If a student sells 20 or more boxes, they get a prize. If they sell less than 30, they get a "small" prize. If they sell more than 30, they get a "large" prize. (Yes you could use an If...Then...ElseIf... statement.)

An example of what this would look like in a specific programming language is:

The flowchart for this type of problem will look something like this:



4.8 Repetition

A loop is a sequence of instructions that are repeated until a certain boolean condition is true (or false). When looping, the program must have an instruction to allow the loop to stop. If the instruction is missing (or present but never triggered), the repetitions will go on forever. This is known as an infinite loop. If you accidentally create an infinite loop, you will have logical error in you program. The program will compile but when you run it, it will most likely give you an overflow error or just hang. The condition that must be triggered can either be at the beginning or the end of the repletion structure. There are several repetion structures, just like there were several different selection structures, each for a different purpose.

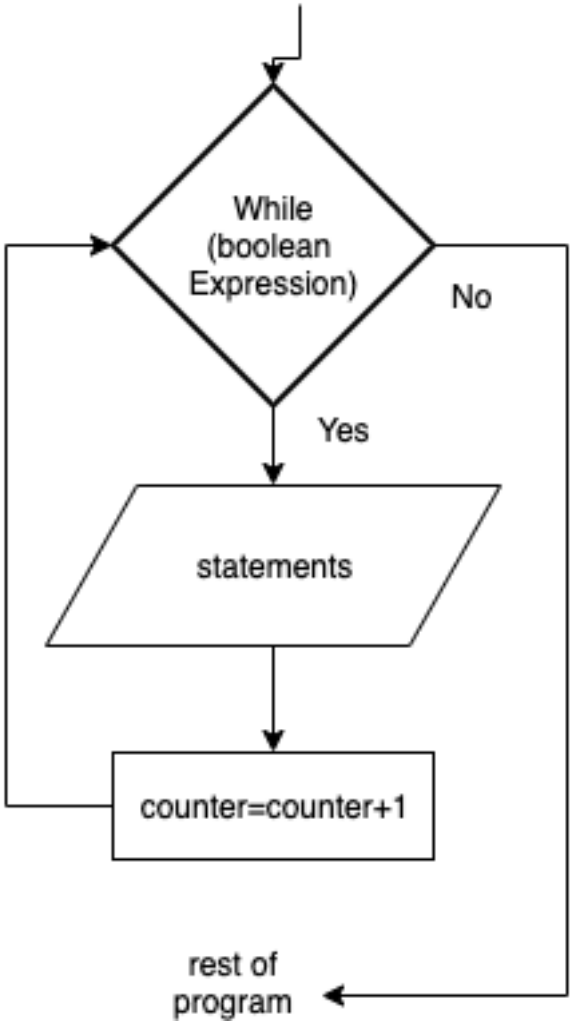
4.8.1 While Loop

The While loop is a repetition structure where the statements in the structure are repeated as long as the boolean expression is true. The flow of logic keeps coming back up to the Boolean expression to check if it is still true. As soon as the boolean expression is false, the flow of logic hops down to the end of the loop. The boolean condition is also checked before the looping statements are executes the first times. This mean if the condition is not true the first time, the statements will never happen.

The while loop (in most computer programming languages) takes the generic form of:

WHILE boolean expression:
    statement(s)
    counter = counter + 1
END

The flowchart for a While Loop will look like this:



It is a common occurrence to have an accumulator or counter within a looping structure. The counter, usually, is incremented (1 is added) or decremented (1 is subtracted) each time the condition is meet and the statements inside the loop are performed. When the counter reaches a certain number that is expressed inside the boolean statement, then the loop is exited. Ensure you use proper style and do not do what is very common in programming, just declare the variable as i, j or x. Always name a variable for what it is actually doing and holding.

The following code snippet, a repetition program. The user enters a positive integer and the program prints out that many lines:

4.8.2 Do While Loop

The Do...While loop is a repetition structure where the statements inside the loop are executed **at least once**. Only then after being executed once, the Boolean expression is evaluated. If the Boolean expression is true, the body of the loop is executed again and the Boolean expression is re-evaluated once again. Note that this is different from the while loop, where the Boolean expression is at the top. Being at the top in a while loop, it is evaluated first and there might be a circumstance where the Boolean expression is false, right from the beginning. In this case the while loop will never happen. In a Do...While loop, the statements will always happen at least once.

The do...while loop (in most computer programming languages) takes the generic form of:

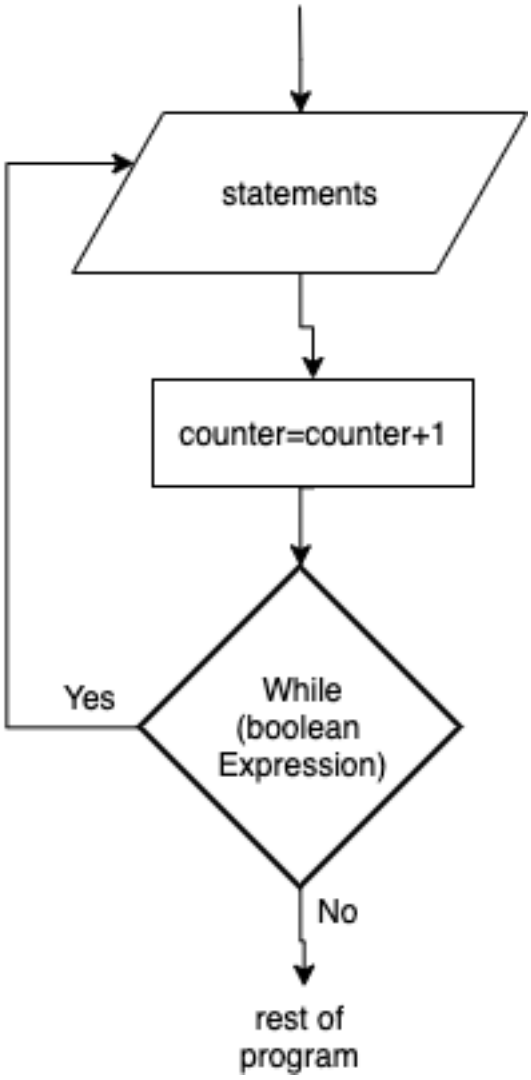
DO

statement(s)

counter = counter + 1

WHILE boolean expression

The flowchart for a Do. . . While Loop will look like this:



The following code snippet is a repetition program. The user enters a positive integer and the program prints out that many lines:

4.8.3 For Loop

The For loop is another repetition structure. The advantage of using the for loop is that the structure itself keeps track of incrimenting the counter, so you do not have to. Usually by default, the counter is just incrimented by 1 each time you go through the loop. As normal, there is some way to exit the loop, usually by some kind of Boolean expression. Some for loops allow you to just specify how many times you would like to do the loop, by providing a number and no Boolean expression.

To use the For. . . Next loop, there will also be a loop counter that keeps track of how many times through the loop the program has executed. Each time the code inside the loop is executed, the loop counter is automatically incremented by 1. Then a expression checks to see that the predetermined number of times has been reached.

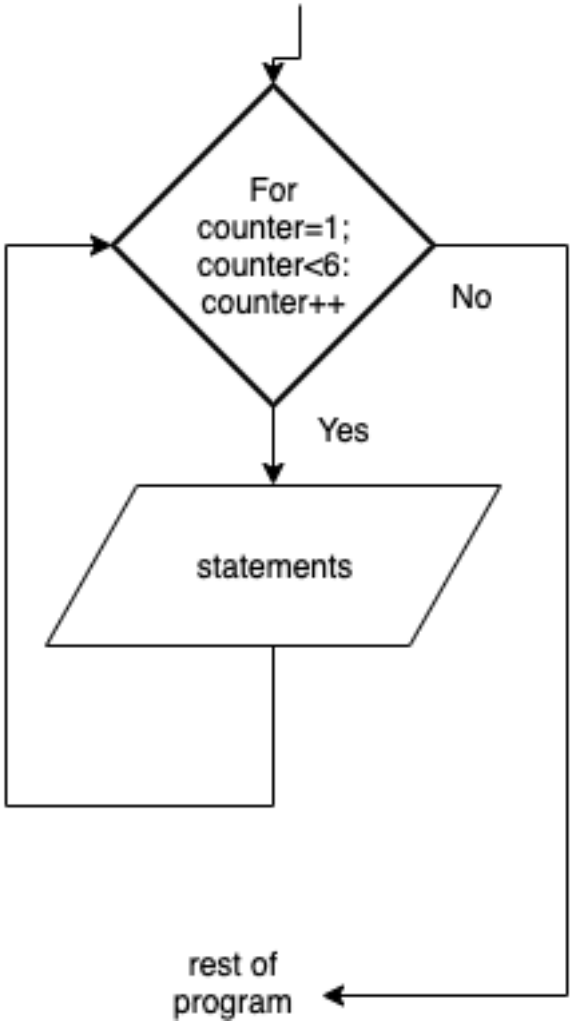
The for loop (in most computer programming languages) takes the generic form of:

FOR counter in range(n)
    statement(s)
END

or

FOR (counter = 0; boolean expresion; counter++)
    statement(s)
END

The flowchart for a For Loop will look like this:



The following code snippet is a repetition program. The user enters a positive integer and the program prints out that many lines:

4.8.4 Break Statements

The break statement can alter the flow of a normal loop. Loops iterate over a block of code until the Boolean expression is false, but sometimes we wish to terminate the iteration or even the whole loop early. The break statement is used in these cases.

The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop. If the break statement is inside a nested loop (loop inside another loop), the break will terminate the innermost loop only. Note you will most likely need to place an if statement inside the loop to use the break statement, because if you just have a break statement all by itself inside a loop, it will always hit it the first time through and that is not really useful!

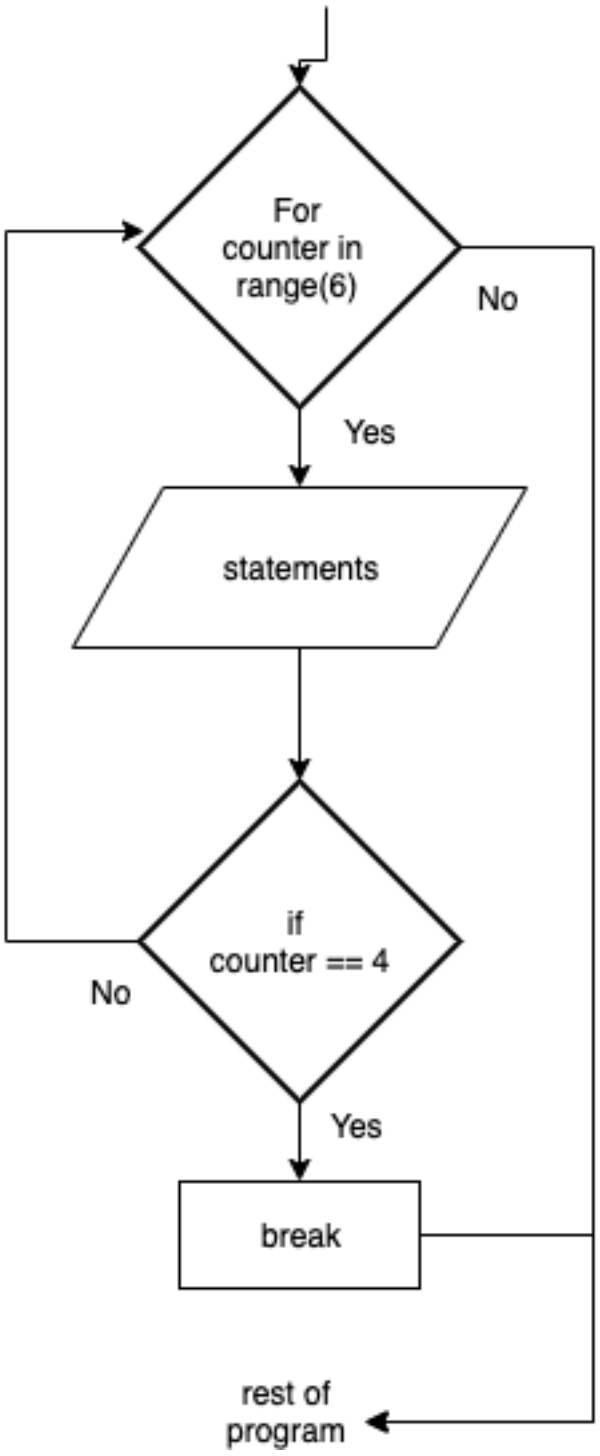
The break statement (in most computer programming languages) takes the generic form of:

WHILE bolean expression
    statement_1
    statement_2

```
    ...
    IF bolean expression THEN
        BREAK
    ENDIF
    counter = counter + 1
END


FOR counter in range(n)
    statement_1
    statement_2
    ...
    IF bolean expression THEN
        BREAK
    ENDIF
END
```

The flowchart for a Break statement will look like this:



The following code snippet is a break program:

4.8.5 Continue Statement

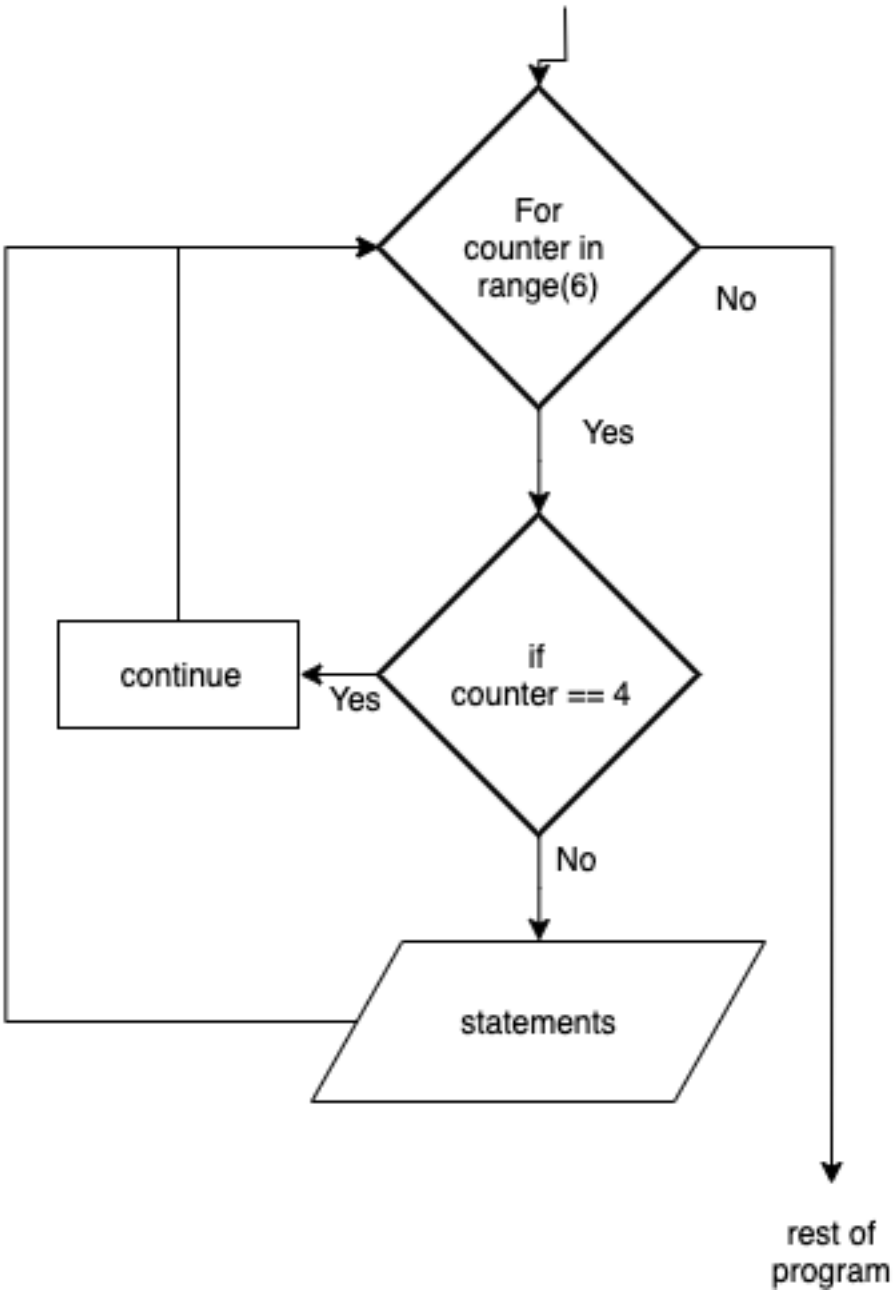
The continue statement gives you the option to skip over the part of a loop where an external condition is triggered, but to go on to complete the rest of the loop. That is, the current iteration of the loop will be disrupted, but the program will return to the top of the loop. The continue statement will be within the block of code under the loop statement, usually after a conditional if statement.

The continue statement (in most computer programming languages) takes the generic form of:

WHILE bolean expression
    counter = counter + 1
    statement_1
    statement_2
    . . .
    IF bolean expression **THEN**
        CONTINUE
    ENDIF
END

FOR counter in range(n)
    counter = counter + 1
    statement_1
    statement_2
    . . .
    IF bolean expression **THEN**
        CONTINUE
    ENDIF
END

The flowchart for a Continue statement will look like this:



The following code snippet is a continue program:

4.8.6 Nested Loops

The placing of one loop inside the body of another loop is called nesting. When you "nest" two loops, the outer loop takes control of the number of complete repetitions of the inner loop. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes.

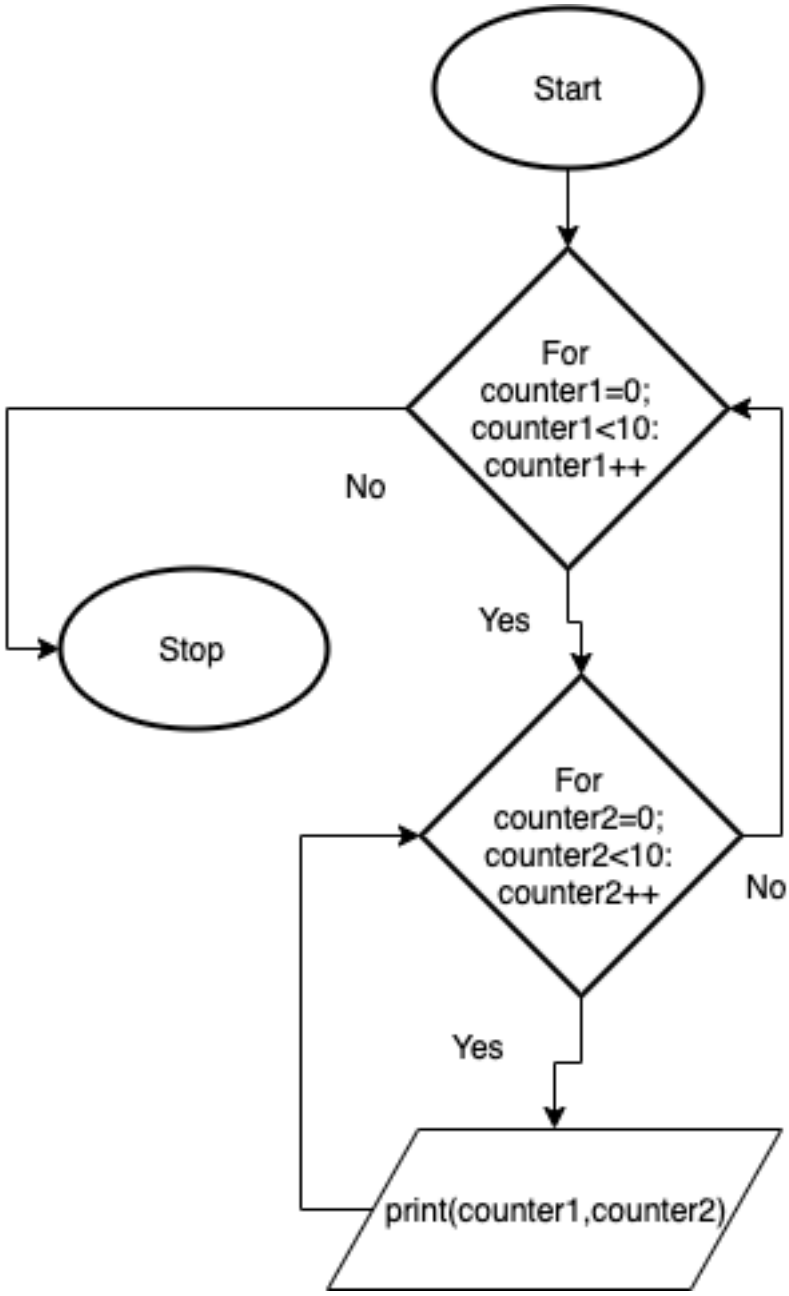
A Nested for loop (in most computer programming languages) takes the generic form of:

```
FOR counter in range(n)
    FOR counter in range(m)
        statement(s)
    END
END
```

or using While loops:

```
WHILE counter1 <= n :
    WHILE counter2 >= m :
        statement(s)
        ...
        counter2 = counter2 + 1
    END
    ...
    counter1 = counter1 + 1
END
```

In a flow chart it looks like:



The following code snippet is a nested loop example of a 2 digit odometer:

4.8.7 Loops and If Statements

As you can probably guess from now, yes you can place loops inside if statements and if statements inside loops.

An if statement inside a loop would (in most computer programming languages) take the generic form of:

FOR counter in range(n)
    IF (boolean expression) **THEN**
        Statements to be performed

```
    ENDIF
END
```

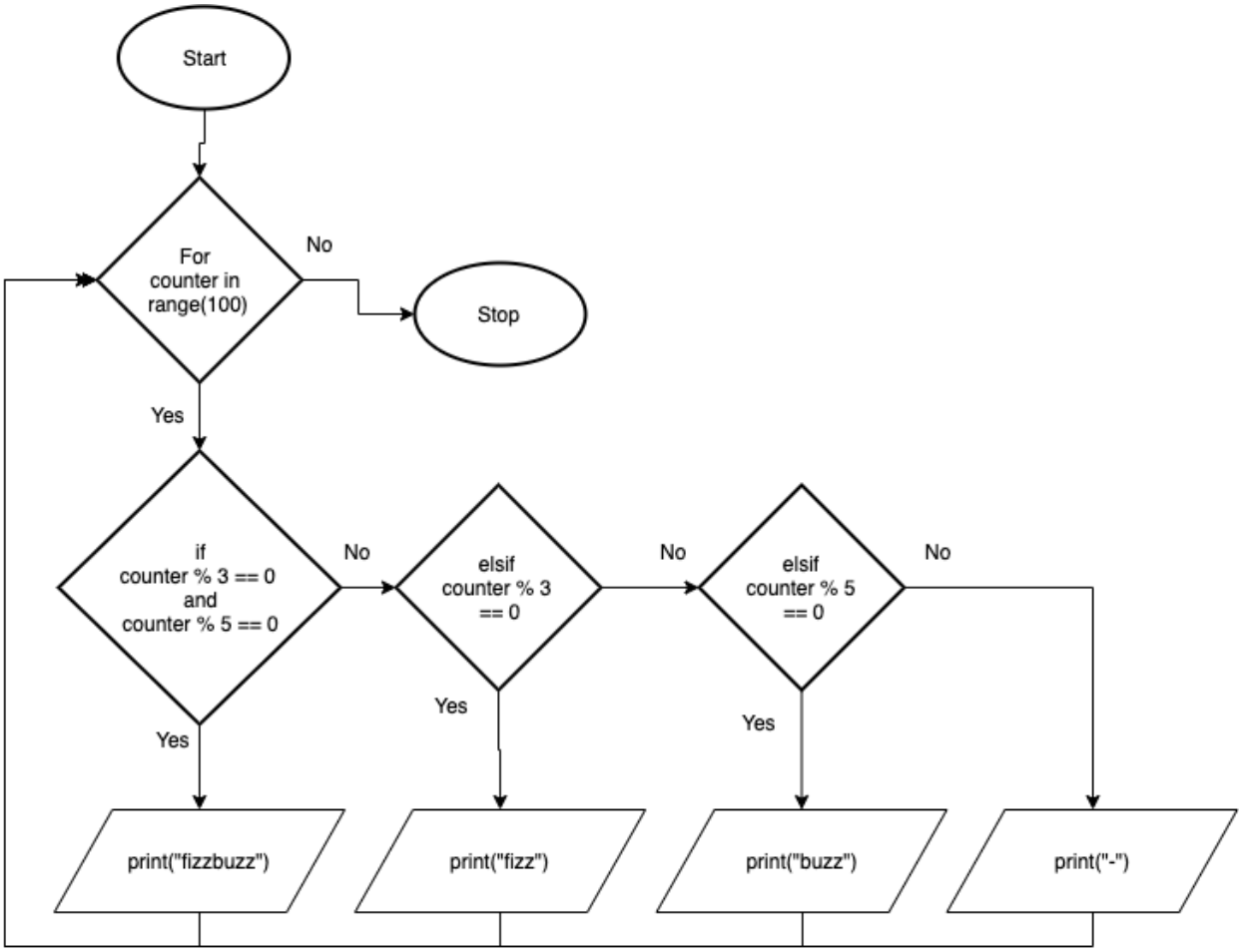
or using While loops:

```
WHILE counter1 <= n :
    IF (boolean expression) THEN
        Statements to be performed
    ENDIF
    ...
    counter1 = counter1 + 1
END
```

Here is one of the most well-known examples of the exercises that you might be given as the opening question in a junior data scientist job interview.

The task is: *Go through all the whole numbers up until 100. Print 'fizz' for every number that's divisible by 3, print 'buzz' for every number divisible by 5, and print 'fizzbuzz' for every number divisible by 3 and by 5! If the number is not divisible either by 3 or 5, print a dash ('-')!*

In a flow chart it looks like:

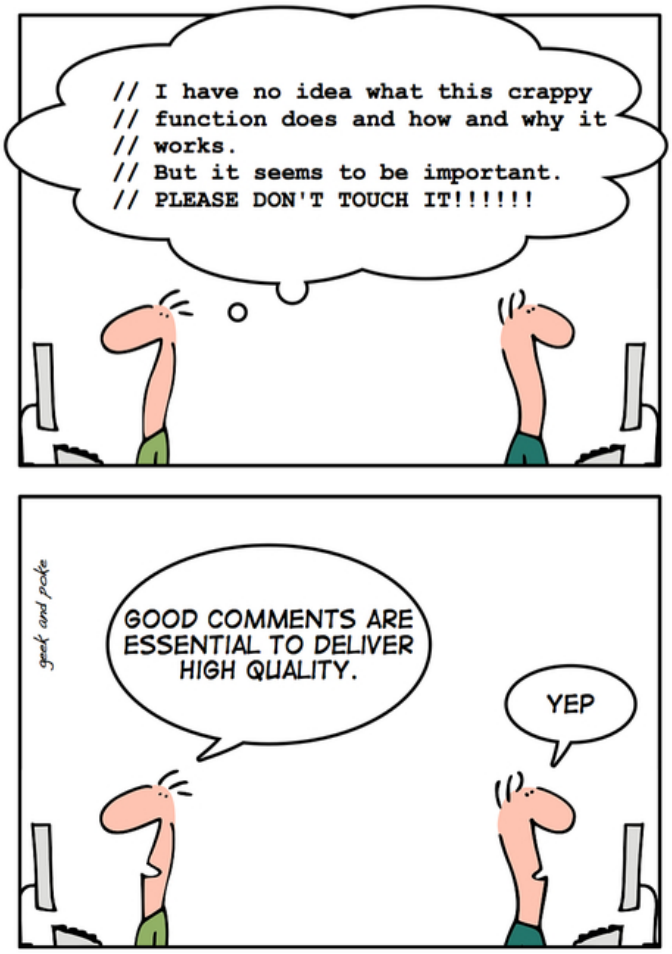


The following code snippet is the solution to the above problem:

CHAPTER

FIVE

FUNCTIONS



The goal of the previous section was to introduce you to the concept of structured programming design. The reasons programs are designed using this method are to:

- easier to develop accurately
- develop in less time
- easier to read and understand
- easier to modify and maintain

- easier to test and correct errors

There are two groups of methods to use in structured programming to ensure the above occur. The first is the three basic control structures we have learned in the previous chapter, sequence, selection and repetition. The second method is modularity. A complicated problem is broken down into smaller problems. Each of these smaller problems is coded. The small sections of code are then organized to make the solution. Modularity in programming is usually done with subroutines or procedures or normally just called functions.

5.1 Understanding Functions

A Subroutine or what we will just refer to as a function is a block of code written to perform a specific task. We have actually been using functions all along our programming journey so far and you probably have not even notices. Many functions are built into most programming languages, like the print() function for example. In python we usually placing our code in main(), which is a function. In C++, it looks for the main() function to start running its program. You can also create functions to do specific calculations, like converting temperature from Celsius to Fahrenheit for example. This type of conversion is very common and we might want to use it in another program (re-usability is very important in programming; why re-invent the wheel, just use someone else's code).

Functions need to have two (2) seperate mechanisms to work correctly. You need a way to create the function in your program and you also need a way to then "call" the function and get the code inside of it to run. Although not normally needed for the computer, we usually place the function definition before we call the function, so that when people are actually reading the code they have seen the function definition and know what it will do *before* they see it being called.

This is a good template for keeping all the parts of your program organized in a program file:

1. comment header
2. global (to the file) constant(s)
3. global (to the file) variable(s)
4. function(s)
5. main body of code

5.1.1 Creating and Calling a Function

Each programming language has its own syntax to create and call a function. Here is an example program that uses functions to calculate area and perimeter (I know there are many ways to improve this code, we will be doing that in the next sections!(:

5.2 Functions with a Parameter

A function often needs pieces of information to be able to complete its work. In the last section you might have noticed that I had to ask for the length and width twice, once for each of the area and perimeter calculations, clearly not ideal. One method of doing this is to declare the variable as a *global variable* and then any code within that file can access it. This is a very bruit force method and is **very bad programming style**. By doing it this way, the variable is created and held in memory for the entire time the file is running, even thought the information may only be needed for a small fraction of the time. Also the function is no longer portable, since it relies on external global variables that might not exist in another program. A better, more elegant and more memory friendly way is to pass the information into the function using a parameter. There are three main ways to pass information to a function, by value, by reference and by object reference. For the time being we will just consider by value. (We will even assume python is doing by value, even though it is really doing by object reference.)

5.2.1 Passing By Value

The first method of transferring information to a function is to pass it "By Value". This means that a copy of the data is made and it is passed over to the function to do with it what it pleases. Since it is a **copy** of the data, any changes to the data are not reflected in the original variable. From the animation below you can see that when the cup is passed "By Value" to the function and then filled up, the original cup is still empty. This is because the variable passed to the function is a copy, not the original variable. If the function changes this variable, nothing happens to the original one.

A variable or value passed along inside a function call is called an parameter . Parameter(s) are usually placed inside a bracket when you invoke the function. For example:

When you are creating your function, you must also tell the program that the function is expecting these two values. To do this after the function name declaration you place in brackets the two declaration statements declaring that the function must be passed in two variable (just like when a regular variable is being declared). If your programming language requires that you declare what type the variables will be normally, you will most like have to do that to.

The following is the function declaration line for the examples above:

Here is a full example of the previous sections program, but now the main function takes care of gettin gthe length and width. This way it only have to ask you the information once and it passes the length and width to each function:

5.3 Return Values

If this was math class, the definition of a mathimatical function might be, "... a function is a relation between sets that associates to every element of a first set exactly one element of the second set.", which is a fancy way to say that if you put in a particular number into the function you get one and only one number out and it will always be the same value. The key is that you actually get something back. In computer science it is also common for functions to give you something back and we normally call this a return value or statement.

We have already seen many built in functions that have a return value, like the square root function:

You will notice that the function is now on the right hand side of an assignment statement and the calculated value is being placed right into another variable. To allow this ability, we normally use the reserved word "return" to pass the value back. In many programming languages, in the definition of the function you must specify what type of variable will be returned. This way the IDE can confirm that the same types are being passe back and placed into variable of the same type. This way the language remains type safe.

Now that we know how to use a return statement, we should no longer print out results inside a function like in the last few chapters. It is much better style to retrun the value from a funstion and let the calling process decide what to do with it. Here is the example from last section, this time using return values:

5.4 Functions with Multiple Parameters

All of the functions that we have looked at to this point, there has been one (1) parameter passed into the function. This is not always the case. There might be cases where you need to pass in two (2) or more peices of infomation. Suppose you have a function that calculates the area of a rectangle. In this case unless you are only going to do squares, you will meed a length and a width.

Fortunately you can pass multiple parameters into a function. The thing to remember is that, since you now have more than one (1) item, the order of the parameters is important, since this is how the computer is going to keep track of the different variables.

Since people are not always great at keeping things in order, many programming languages (but not all, for example C++ does not do this) let you pass multiple parameters to functions using "parameteres by keyword". This means that

you actually give each parameter a name and then refer to this name when you are passing the values to the function, so there is no confusion about what value is going where.

In the example below, I have a function that can calculate the quadratic formula. Is is important to keep all three (3) parameters organzied, or you will not get the correct answer. To do this each parameter will be given a name:

5.5 Default Values

All of the functions that we have looked at to this point, you had to ensure that you were sending the exact same number of parameters to the function as it was expecting. To help us do this a good IDE will have "auto-complete" gives us a little pop out window to show us what should be passed over to the function.

Some built in functions we have been using can be accessed in multiple different ways though. For example in Python there is a built in function called random.randrange(). It is kind of like random.ranint() that we have used in the past. Here is the definition for random.ranint():

```
random.randint(a, b)
// Return a random integer N such that a <= N <= b. Alias for randrange(a, b+1).
```

Notice that "a & b" are our starting and ending points.

Here is the definition for random.randrange():

```
random.randrange(start, stop[, step])
// The positional argument pattern matches that of range().
```

First off there is actually 2 seperate ways we could call this function:

- random.randrange(start, stop)
- random.randrange(start, stop, step)

It seems that step is "optional", which it is. By default, if you do not provide it, then python assumes the value is just 1. You can choose for example to place a in 2, and then only even numbers will be chosen. Here is how we would define the function random.randrange() to get this optional parameter:

```
def randrange(start, stop, step = 1):
```

Notice that right in the declaration of the function, the "default optional parameter" is being set. If it is not provided as a parameter, the default value is just used. Each programming language has its own syntax to make this kind of optional parameter work. Here is an example:

5.6 By Value or By Reference

The second method of transferring information to a function is to pass it By Reference. This means that a pointer or *reference* to where the data is stored in memory is passed to the function and not a copy of the data. Since a pointer to where the data exists has been passed, if you actually change the value of the data in the function, the actual values of the data in the main program where the function was called from will also be changed. This can be very powerful but also **very dangerous**. Be careful using passing parameters By Reference, you might mistakenly change a value when that is not what you expect. The rule of thumb is that unless there is a really good reason to pass something By Reference, you never do and you always pass parameters By Value (even though it takes up more space in memory).

Each language has its own syntax on how to declare you are going to accept a value by reference when you are declaring a function he is an example:

5.7 Recursion

Up to this point we have been focusing on a structured approach to programming. We have ensured that we can easily follow the flow of a program, even if we create functions to modulate out components that we use often. There is one more interesting way to solve programs, that uses modularity but in a strange method. We have become accustomed to calling a function, doing the action and then returning to the flow of the program. But what would happen if in a function we called the *exact same function?* If we do this it is called recursion.

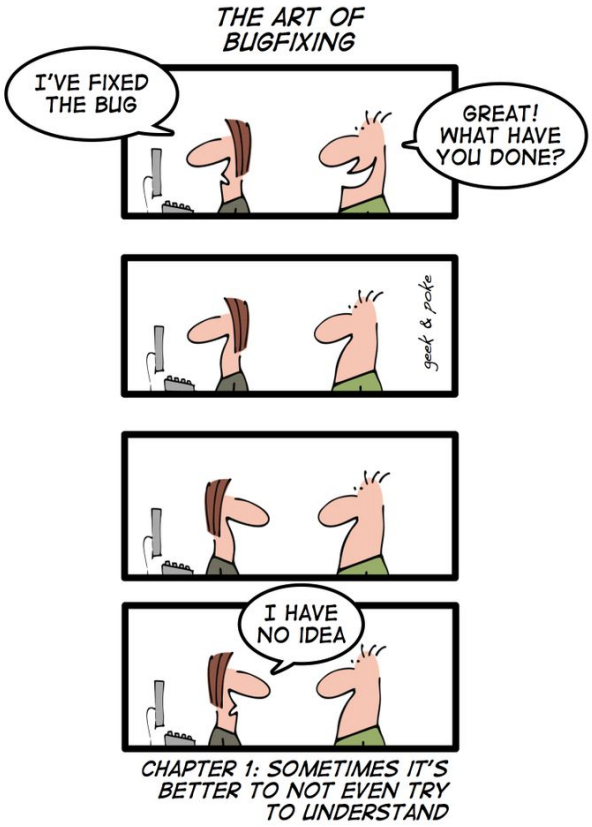
The first question we need to ask ourselves is can we legally do this. This is a really good question, since in some programming languages, recursion is actually illegal and you will not be able to compile you program. Most modern languages actually let you do this. The second question you might be asking is, would I end up in an infinite loop? The answer is you might, just like you could in any looping structure if you do not write the code correctly. But if you write the code "properly" you will not end up in an infinite loop.

By using recursion, what you are trying to do is simplify out the problem to something very trivial to program. Remember that in our problem solving model, trying to simplify out the problem was really important. Here is an example program that reverses the characters in a string, an easy enough task with a loop but by using recursion, it become even easier.

CHAPTER

SIX

HOLDING DATA

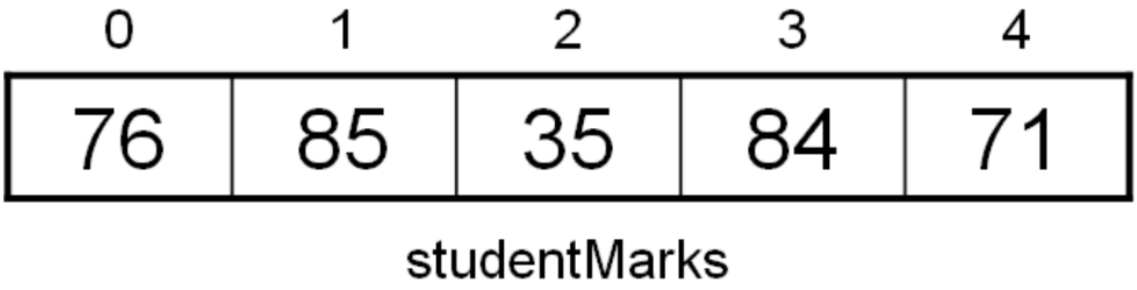


Up to this point, every time we saved a piece of information (a number, string, an object ...) into a variable, it has always been a single piece of information and we have saved it into one single variable. This makes good sense for many things but sometimes it is very inconvenient. If I asked you to save the final mark for every student in this class, we would have to create a variable for every student in this class to hold their mark. Recreating these variables over and over again, should be the clue that there is a better way of doing this. Another problem is the program would only be useful for a class that had that *exact* same number of students, not very likely to occur often. We might also want to save different peices of information about the students, like grade, courses taken. Each of these peices of information are related to a single student, how would we handle that kind of situation?

To solve these problems, there are several different type of data structures we can use, besides just a single variable like we have been using up to now.

6.1 Arrays

An array stores many pieces of data but in the same variable. For example I could save the marks for 5 students in an array like:



This array has 5 elements (note that you usually start counting at 0 with arrays!) but they all have just one variable name (studentMarks). To refer to a specific mark you place the index of the mark after the variable name, usually in brackets. For example, you would refer to the mark of 84 as:

Arrays are an important programming concept because they allow a collection of related objects to be stored within a single variable. To declare an array, you usually must specify how many elements will be in the array during the declaration. Here we are declaring the variable studentMarks and allowing 5 items in it:

This will create our student mark array and ensure 5 student marks can be held. We often use a loop to either place infomation in an array or to get the infomation out of an array, since we need to do the same process for each element in the array. Here is an example:

6.1.1 Array as a Parameter

We know from the previous section that functions are a great way to ensure that your program is modular in its design. Any time a piece of code needs to be repeated more than once or twice, a function or might want to be used. When we use functions, we passed variables by value (making a copy) or by reference (passing the pointer); so that the function can do some process on the data. We normally pass variables like integers, strings, and floats but we have seen that you can pass any object, like an image. Since an array is just a variable that happens to hold several values and not just one, it also can be passed to a function, either by value or by reference.

There is some disagreement in the computer world whether it is wise to pass arrays, especially large ones with many values in them, by value. This is because you are making a complete copy of the array and it could take up a large quantity of memory. Other programmers do not like the idea of passing by reference if you do not want the original array to change, because there is always the risk that you or someone that comes after you, will change the array by accident. They argue that modern computers have so much memory these days (as compared to the "old days") that the risk of changing the original array is not worth the potential memory usage. We will continue to pass variables into parameters by value, unless there is a really good reason that you want to pass the object in by reference.

To pass an array into a function, you declare the array inside the name of your function, just like you have been doing for regular variable, but you **do not place** any brackets or a number inside brackets, like when you were declaring your array normally. This is because the function does not know exactly how many elements the array will have. If you had to set it to a fixed amount, your function would not be very flexible. When the array is passed into the proc function, it will determine how many elements it has and an appropriate array variable with that many elements will be created.

To declare an array as a parameter in a function, it would look like this:

To pass an array into this function as a parameter, it would look like this:

Here is a complete example of creating an array and passing it as a parameter to a function:

6.1.2 Arrays as a Return Value

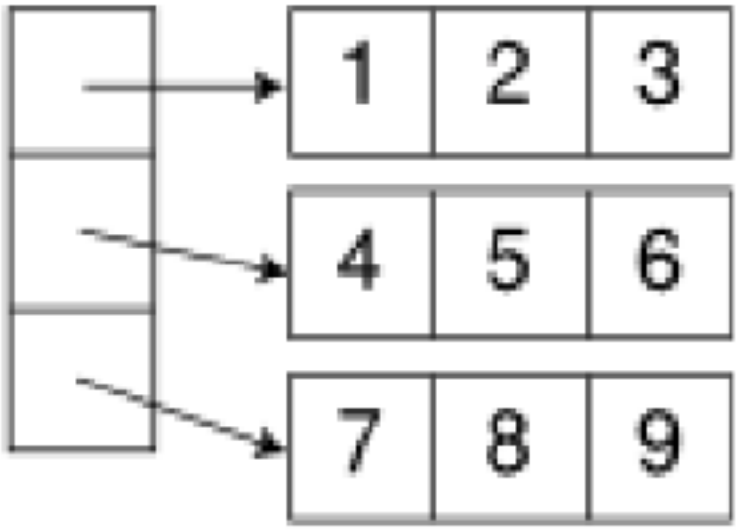
6.1.3 Arrays and For . . . Each Loops

If you think way back to when we did different types of looping structures, one of the methods to loop was using the For loop. It turnes out that since an array is a collection of variables held in a common structure, you can use a for loop with it. This type of loop, usually called a For . . . Each loop, is used when you have a collection of things and you wanted to iterate through each one of them, one at a time. From the previous example of summing up all the values in an array, a For Each loop would look like the following function:

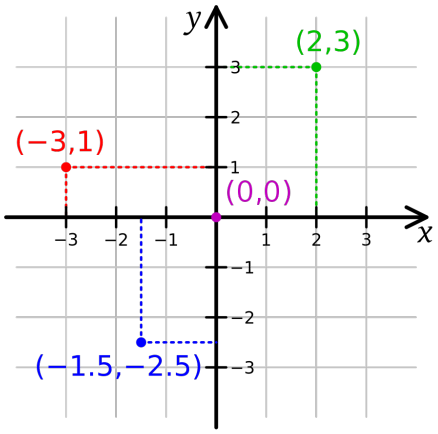
6.1.4 2D Arrays

All the arrays that we have used thus far have been to represent a collection of information. This is a very powerful tool and can save the programmer a lot of time and confusion when dealing with items that are somehow related to each other. Not all things can be represented with a single collection though. Several times we use a grid or spreadsheet to keep information in rows and columns. This matrix of information can not be represented in a single array. In these situations we represent our data with a 2-dimensional (or multi-dimensional array).

A 2-D array can just be thought of an array of arrays.



We represent a given element with 2 indices now, instead of 1 when we had a single dimension. Unlike in math class where you used the Cartesian plane, and moved in the X direction and then the Y direction,



in computer science you move up and down in the rows first and then across to the column position. Thus if we want to refer to the element in the above array that has a value of 8, we would say, studentMarks(2, 1).

There are many applications of 2-D arrays, like a game board (tic-tac-toe), adventure games and business applications like spreadsheets.

6.2 Lists

Since in many programming languages, when you create an array the size must be set during coding so that the memory can be allocated when it is compiled, there is no way to dynamically change the size of an array during run time. This can be a huge disadvantage. One great example of wanting to have dynamic arrays is the classic video game, Space Invaders. If you imagine that all the lazer you shoot are held in an array, then how big should the array be? You have no idea how fast the user can press the "A" button. You can not tell how many lazers might be on the screen at any given time! If you cannot change the size of the array, what can we do, just making a "huge" array is wasteful and not really practical. Fortunately in many programming languages there is a class called "Lists". A list is similar to an array in that it is an ordered grouping of data. You still reference the items in the list using an index. The key difference is that the size of the list can shrink and grow, during run time as needed. As you need to add items, you just use an ".Add" method (or something similar like append). The list class usually has many useful methods for adding, sorting, clearing, finding the length an so on.

Here is an example of creating a list of items:

6.3 Tuple

6.4 Associative Array

An associative array, dictionary, map, or symbol table all refer to the same thing, usually, in computer science. It is an abstract data type composed of a collection of key and value pairs, such that each possible key appears at most once in the collection. The key and value can usually be of almost any type, although usually the key is represented by a string. (If it was an integer, then you would just have an array!). The data type usually comes with operators like, the addition of a pair to the collection, the removal of a pair from the collection, the modification of an existing pair, the lookup of a value associated with a particular key.

You can think of associative arrays like a list of phone numbers. In this list, you can look up a person's name by finding their phone number. The name is the value and the number is the key. This list would look like the following

table:

Phone Number	Name
1-203-456-4657	John Doe
1-964-725-5617	Jane Smith
1-275-486-8562	Daniel Brown
1-347-374-3412	John Doe

Associative arrays have two important properties. Every key can only appear once, just like every phone number can only appear once in a directory. And, every key can only have one value, just like every phone number can only refer to one person. It is possible that two people can have the same name in this list, so it's important to remember that a given value can appear more than once in an associative array.

6.5 Sets

6.6 Stacks

6.7 Queue

6.8 Heap

6.9 Graphs

6.10 Binary Trees

CHAPTER

SEVEN

USING OOP

...

CHAPTER

EIGHT

CREATING OBJECTS

...