

---

# PYTHON INTRO LABS

---

CMPUT 274/275

2019-2020



*Veronica Salm  
Department of Computing Science  
University of Alberta*

---

# TABLE OF CONTENTS

---

<b>1</b>	<b>How to Get the Most Out of the Intro Labs</b>	<b>3</b>
<b>2</b>	<b>What is Python?</b>	<b>4</b>
2.1	Introduction to Computing Science . . . . .	4
2.2	Why Python? . . . . .	5
2.3	Getting Started . . . . .	5
<b>3</b>	<b>Lab 1 - Hello World</b>	<b>6</b>
3.1	The Interactive Python Interpreter . . . . .	6
3.2	Variables in Python . . . . .	8
3.3	Running Your Own Python Files . . . . .	9
3.4	Making a Python Program . . . . .	10
3.5	Your Tasks . . . . .	11
<b>4</b>	<b>Lab 2 - Using the Terminal</b>	<b>15</b>
4.1	Introduction . . . . .	15
4.2	Terminology: Terminal, Shell, Command Prompt . . . . .	15
4.3	Essential Linux Basics . . . . .	16
4.4	Useful Commands . . . . .	20
4.5	Ways to Learn More . . . . .	26
<b>5</b>	<b>Lab 3 - Dice Simulator with Outcomes</b>	<b>29</b>
5.1	The Random Module . . . . .	29
5.2	Practice in the Interpreter . . . . .	30
5.3	Version 1: Setting Up the Dice Roller . . . . .	32

---

5.4	Conditional Statements (if/elif/else) . . . . .	32
5.5	If Statements . . . . .	34
5.6	Version 2: Adding Outcomes to the Dice Roller . . . . .	36
5.7	Your Task: Finish Version 2 . . . . .	37
5.8	Issues With Version 2 of the Dice Roller . . . . .	37
5.9	Lists . . . . .	38
5.10	Your Task: Create Version 3 . . . . .	39
5.11	Why all these versions? . . . . .	39
<b>6</b>	<b>Lab 4 - Preparing for Morning Problems: Input and Built-In Python Functions</b>	<b>41</b>
6.1	Input: Strings . . . . .	41
6.2	Input: Integers or Floats . . . . .	44
6.3	Input: Lists . . . . .	47
6.4	Output Tricks . . . . .	51
6.5	Your Task: Date Converter . . . . .	53
<b>7</b>	<b>Lab 5 - Iteration: Clocks, Fast and Slow</b>	<b>54</b>
7.1	Clocks, Fast and Slow . . . . .	54
7.2	Loops . . . . .	54
7.3	Modular Arithmetic and the Modulo Operator . . . . .	59
7.4	Your Tasks: Loops and the Modulo Operator . . . . .	60
7.5	Nested Loops . . . . .	62
7.6	Your Task: Solving the Fast and Slow Clocks Problem . . . . .	64
<b>8</b>	<b>Lab 6 - Problem-Solving and Algorithms</b>	<b>66</b>
8.1	What is an Algorithm? . . . . .	66
8.2	Devising a Strategy to Solve Your Problem . . . . .	68
8.3	How to Test and Debug: The Basics . . . . .	70
8.4	Problems to Solve . . . . .	74
8.5	Links to External Problems on Kattis . . . . .	79

## SECTION 1

---

# HOW TO GET THE MOST OUT OF THE INTRO LABS

---

This collection of introductory labs is designed to give students a head start on learning the fundamentals of Linux terminal navigation, the Python programming language, and other key skills for CMPUT 274/275. They are aimed at beginners, but are designed to help advanced programmers gain valuable practice as well.

A guide to using this document:

1. **Beginner:** Read the book at your own pace from start to finish. The progression is designed to teach you all the basic skills you will need if you have never programmed or used a terminal before. Expect to have to put in some extra work learning everything you will need to know.
2. **Intermediate/Advanced:** If you already have some programming experience, simply skip to every section labelled “Your Task”. These are the practical programming exercises that will test your knowledge of each section. If you can do the problems without issue, you likely do not need to read the full section. If you have any problems, jump back and do the reading.
3. Finally, all students, regardless of experience level, should **make sure to read the Style and Code Submission Guidelines Document** (which can be found on eClass or [here](#)). Proper style guidelines may be new to you and will be something you are graded on in CMPUT 274/275.

Here are a few additional tips to help you get the most out of these labs to prepare for CMPUT 274/275:

- As a computing scientist or computer engineer, you will be expected to learn how things work on your own. Be prepared to do research when you do not understand something. Work to become skilled at searching the internet for documentation and solutions to common problems and errors you will face.
- There are many hyperlinks in this pdf. Click them and read through them, especially if you are struggling with a concept. They will help you gain a more thorough understanding of each data type and structure, or may explain things in a way that makes more sense to you.
- Do every task. Don’t just read through them, **do them**. You will learn more by reading and then doing than by reading alone.

## SECTION 2

---

### WHAT IS PYTHON?

---

“Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later.”

- From [python.org](https://python.org)

#### Learning Outcomes:

- Introduce computing science and motivate the Python language.

### 2.1 INTRODUCTION TO COMPUTING SCIENCE

[Computing science](#) is a discipline that spans both theory and practice. Computing scientists are interested in the science of problem solving. They must be adept at understanding problems by modeling, analyzing, and decomposing them into manageable pieces. They must be able to design solutions and verify that their solutions are correct. This requires precision, creativity, and careful reasoning.

The goal of a computing scientist is to understand the “why” behind computer programs. They aim to use advanced mathematics and algorithms to discover new ways of manipulating and transferring information.

As a computing scientist or computer engineer, you will solve problems by first understanding them, then decomposing them into manageable pieces, and finally implementing (or perhaps creating!) any necessary [algorithms](#) or [data structures](#).

You can use [Python](#), a [programming language](#), to specify instructions to solve a problem and tell your computer to produce a certain kind of output. Python is a tool that can be used to solve the sorts of problems you will encounter as a computing scientist.

#### Learn more:

- [What is Computer Science?](#)
- [Difference Between a Computer Science and Information Technology Degree.](#)

---

## 2.2 WHY PYTHON?

Python is a general purpose, high-level programming language. It has clear syntax, making it simple to read and understand compared to low-level languages like C or C++. It can be used to quickly and easily introduce many programming concepts.

Python is also a powerful language with the support of many built-in functions, modules, and libraries. This functionality allows easy access to everything from data display to web development to creating computer games.

## 2.3 GETTING STARTED

Before we start learning to program in Python, there are two useful things you should know about the language.

Firstly, Python comes in two different flavours: Python 2 (commonly referred to as “Python”) and Python 3. They are not the same thing. Although Python 3 is newer and Python 2 is being phased out, both are still commonly used. In this course, we will teach Python 3. This is an important distinction because a search for help with “python” will typically result in help with Python 2 (which can be very confusing!). Make sure to always add the tag “python3” to your search and check to make sure the article you are reading specifies Python 3.

We will not discuss the differences between Python 2 and Python 3 in this course. If you are curious to learn more, these articles may be of interest:

- [Python 2 vs Python 3: Practical Considerations](#)
- [Should I use Python 2 or Python 3 for my development activity?](#) from python.org.

Secondly, the [Python documentation](#) is your best friend. Seriously. Although it can seem confusing at first, stick with it. Once you get used to reading documentation, you will find that it contains all the information you need to know about how to use any built-in Python functionality.

With that out of the way, it's time to learn how to program in Python. Let's get started!

## SECTION 3

---

### LAB 1 - HELLO WORLD

---

This section will teach you how to write, save, and run Python programs. There are multiple ways of running a Python program: using the interactive interpreter prompt, running a source file, or [using an IDE](#). This section will demonstrate how to use the first two methods.

#### Learning Outcomes:

- Be able to launch the Python interpreter and write simple code using its interface.
- Explore the `print()` function.
- Know how to save your code in `.py` files and run them using the terminal.
- Understand single-line comments.
- Gain experience working with and understanding variables and code written by someone else.
- Become familiar with some error messages generated by Python.

### 3.1 THE INTERACTIVE PYTHON INTERPRETER

The Python interpreter can be used from an interactive shell. This can be very handy for testing code snippets in a safe environment.

The interactive shell waits for commands from the user, executes each command, and returns the result of the execution. After this, the shell waits for the next input.

#### 3.1.1 LAUNCHING THE INTERPRETER

This tutorial assumes you are working in the CMPUT 274 Virtual Machine. If you are not, please follow the instructions on the eClass page to install and open it before returning here.

Open a new terminal. You can do this by clicking the Terminal icon on the left sidebar (it should look like a black box containing the characters `>_`). If the Terminal icon is not pinned to the sidebar, you can search for it by clicking the “Search your computer” icon in the top left corner and searching for “Terminal”.

This should open a terminal to your home directory. You should see a prompt that ends with the characters `~$`.



3.1: The terminal icon.

---

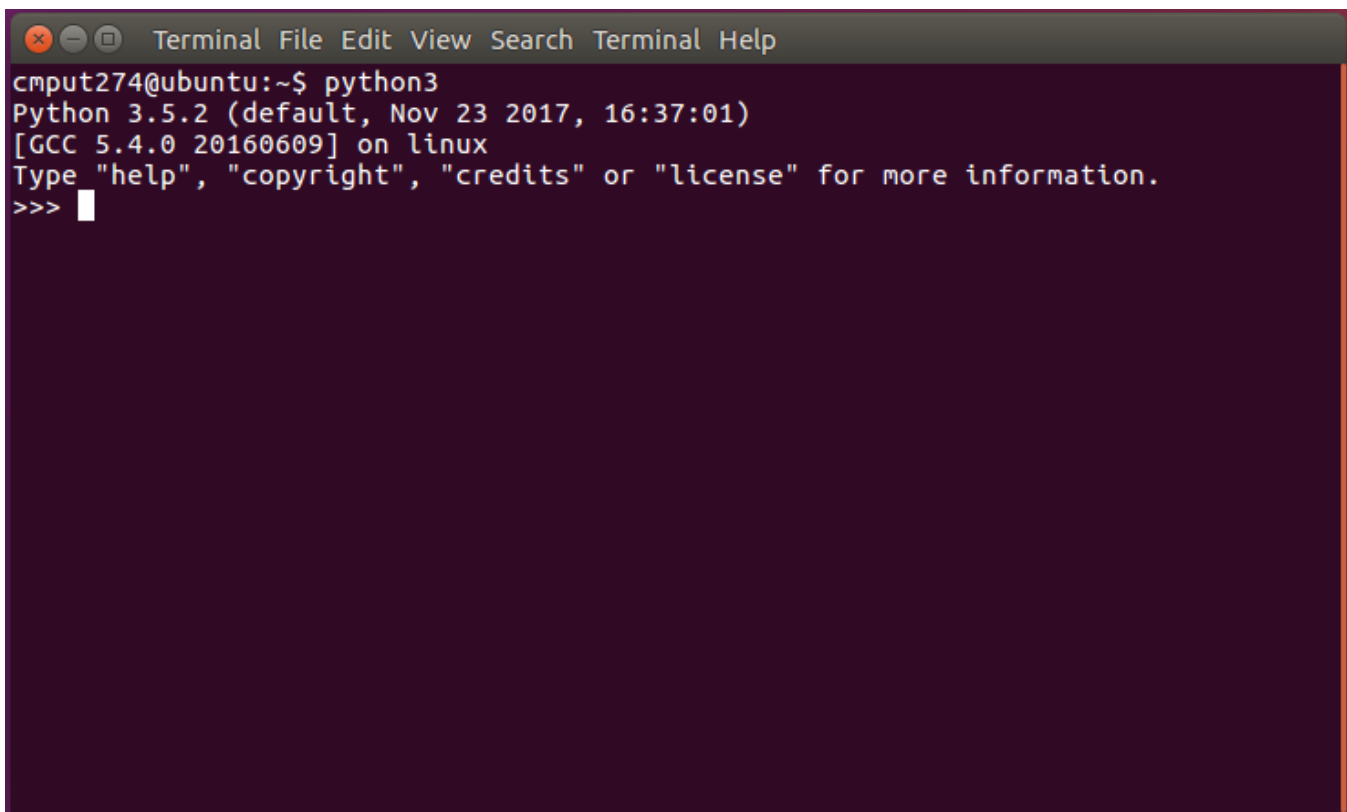
Now, in the terminal, simply type:

```
python3
```

into the terminal prompt and hit **Enter**. You should now enter the Python interpreter. Make sure to type `python3` and not `python`, to avoid accidentally launching Python 2 instead!

**Note: Instead of typing `python3`, you can type `ipython3` instead.** This stands for interactive Python, and will take you to an environment with syntax highlighting (meaning that strings, function names, and types will be highlighted in specific colours). This interpreter functions the same way as the official interpreter, but can be easier to code with because of the colour highlighting.

You should see a window that looks like this when the interpreter is successfully launched:

A screenshot of a terminal window titled "Terminal" with a menu bar containing "Terminal", "File", "Edit", "View", "Search", and "Help". The terminal shows the command "python3" being executed at the prompt "cmput274@ubuntu:~\$". The output displays the Python version "Python 3.5.2 (default, Nov 23 2017, 16:37:01)", the compiler "[GCC 5.4.0 20160609] on linux", and instructions to type "help", "copyright", "credits", or "license" for more information. The prompt ">>>" is shown with a cursor.

### 3.1.2 USING THE INTERPRETER: HELLO WORLD!

You can now enter some code for Python to run. Try:

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

You should see the string “Hello World!” printed in the terminal. `print()` is the function Python uses to display program output to the user. “Hello World!” is a [string literal](#) that Python has just printed. By default, `print` terminates the line with a [newline character](#).



---

### 3.1.3 EXPLORING THE INTERPRETER

Now that you have the interpreter open, you can use it to evaluate simple Python expressions and statements.

The interpreter is a safe environment where you can try Python statements to see how they work. This can be extremely useful, especially when learning new functions or libraries. Anytime you don't know how something works in Python, the answer is to **just try it**. Type it into the interpreter and see what happens!

Try all of the following:

- **Basic math:**

```
>>> 3 + 5
>>> 7 * 4
>>> 21 - 5
>>> 3 ** 4
```

You should see the answers appear one by one. The interpreter evaluates each line as you type it, and echos the answer back to you. What did each operator do? Try other arithmetic expressions.

- **Experiment with `print()` and [take a look at its documentation](#).** Try the following lines:

```
>>> print('CMPUT "274"')
>>> print("Hello", "World")
>>> print('H', 'E', 'L', 'L', 'O')
>>> print(5+7)
>>> print("This is a mistake')
```

You should see an error message after the last line executes. Note that a string literal like “This is a mistake” must have the same type quotation marks (in this case, ” and not ') at both the start and end.

- **Type `help` to see what happens.**

- In the interactive help menu, ask Python about the `print` function using `help(print)`. How does the help menu's answer compare to the Python documentation? If you are curious, look up a few more keywords, topics, or terms you don't know, such as `help(len)` or `help(max)`. You can type `q` to leave the help window if you are stuck.

- **Find out [how to leave the Python 3 interpreter](#).** Experiment! If you can't figure it out, search the internet to find the answer. There are at least three different ways to do it (Note: closing the entire terminal doesn't count!).

## 3.2 VARIABLES IN PYTHON

So far, the Python interpreter has been convenient for quick calculations, but what if we want to store a calculation so that we can quickly and easily use it again?

---

To do this, we need to introduce the idea of a variable. A variable is an item of data in a program. Most variables have **names** (or identifiers). Every variable has a **value**. As the program runs the value of a variable can change. Variables can store values that have a specific kind of type, for example integer or string.

In Python, an **assignment statement** is used to assign an expression to an identifier in the following way:

```
(the variable name) = (the value you want to store)
```

For example:

```
>>> a = 4+5
>>> b = 7
>>> c = a+b
```

are all examples of assignment statements.

By storing the output of  $4+5$  in  $a$  and  $7$  in  $b$ , we could easily access those values and use them to create  $c$ . In the interpreter, you can type the name of a variable to see what it contains. Try typing  $c$  and then hit **ENTER** to see what value it stores.

In Python, you can attach almost any value to an identifier. Try the following lines of code in the interpreter:

```
>>> first_name = "John"
>>> last_name = "Doe"
>>> full_name = first_name + last_name
```

Try typing `full_name` to see what it contains. This is an example of [string concatenation](#), which we will explore in more detail later. Here is another example:

```
>>> n = 5.7
>>> n+1
>>> print(n)
>>> n = n+2
>>> print(n)
```

The variable  $n$  contains a new type of value, a [literal float](#). Notice how the variable only changes when a new value is assigned to the identifier  $n$  using the assignment operator `=`.

## 3.3 RUNNING YOUR OWN PYTHON FILES

The interpreter is useful for quick calculations and short programs. However, for longer programs it is useful to write the code in a file and run the entire file (which can have dozens or hundreds of lines of code) all at once. This allows you to both save code for later and work on much larger projects.

### 3.3.1 NAVIGATING TO A NEW FOLDER

Because you want to save Python files, you should now move away from your home directory and create a dedicated directory (or folder) for storing your projects. Type the following lines into your

---

terminal prompt to move to your Documents folder, create a new folder, and move into this folder.

```
mkdir ~/Documents/Python-Intro-Labs
cd ~/Documents/Python-Intro-Labs
```

**Note:** If you receive a `NameError` or something similar after running one of these commands, it is likely that you are still in the python interpreter. Run `quit()` to exit the interpreter and then try running the commands again.

What did this do?

- `mkdir` creates a new directory with the name/path provided. If you type `mkdir ~/dirname` then it creates that directory in your home directory. So in this case, a new folder called “Python-Intro-Labs” was created within your Documents folder.
- `cd` changes your current directory to the provided path, so now you are located within the new directory you just created.

For more on terminal navigation, [click here](#) to read the section on Using the Terminal.

The name “Python-Intro-Labs” can be changed to create whatever name you wish to use for your new folder.

Congratulations! You are now located in the directory you just created. You should see that its path is now included in the prompt. In the future you can skip the first command and simply type:

```
cd ~/Documents/Python-Intro-Labs
```

to jump to that path directly.

## 3.4 MAKING A PYTHON PROGRAM

Now that you have created a new folder, it’s time to create and run your first Python file. We will do this using a [text editor](#) called [Sublime Text](#), which is installed on the VM.

Type:

```
subl helloworld.py
```

into the terminal and press enter to create a new file.

This should also open a Sublime Text window in which you can edit the file. The `.py` [extension](#) indicates that this file will contain Python code.

In the file, type the line:

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

Make sure to save the file. You can do this with `CTRL+S` or `COMMAND+S` on a Mac. Alternately, you should be able to save the file using the file menu and clicking `File->Save`.

Once the file is saved, switch back to the terminal. You can run a Python 3 file using the following general syntax:

---

```
python3 filename.py
```

where `filename` is the name of the file you wish to run.

If you successfully run your file (`helloworld.py`), you should see the string “Hello World!” printed in the terminal (though you should not see any quotes). Congratulations! You just ran your first Python program.

### 3.4.1 COMMENTS

Sometimes, it can be useful to add some lines explaining the meaning or purpose of your code. In Python, a comment can be a single line starting with a `#` character:

```
# This is a comment.
```

or can run for multiple lines if enclosed by triple quotation marks:

```
"""
This is also a comment.
On multiple lines.
"""
```

Comments are deliberately ignored by the interpreter when your Python code is run - even when they contain Python code. It is strongly encouraged to use comments to make your code more understandable to other humans who may read your code. In CMPUT 274/5, you **must** use comments to supplement any vague, complex, or difficult code you write.

## 3.5 YOUR TASKS

### 3.5.1 TASK ONE: SCARF.PY

You will have two major tasks in this lab. The first is an exercise working with variables and comments, in which you will design a scarf.

The program will use the code below, which you can [copy from this link](#). Do not copy the code from the PDF document - it will change the formatting and result in errors.

```
#-----
# Change these variables to design your own scarf!
# Add a comment describing what each variable does.
#-----

# this variable...
colours = ['#', '|']

# this variable...
colour_length = 1

# this variable...
```

```

pattern_length = 25

# this variable...
pattern_width = 10

#-----
# Don't change anything below this line!
#-----

print("Here is your scarf:\n")
for pos in range(int(pattern_width * pattern_length)):
    print( colours[ int((pos)/colour_length) % len(colours)], end="")
    if (pos % pattern_width) == pattern_width-1:
        print("")

```

Once you have the code in your clipboard, open a new file, name it `scarf.py`, and paste in the code. If you are still in your target directory, recall that you can do this using:

```
subl -n scarf.py
```

Now follow the instructions in the program: make changes to the variables and see what happens. Add a comment describing what each one does. **Make sure you save your code each time, or you will not be able to see your changes!**

**Note:** The identifier `colours` stores a type of object you have not yet seen: a *list*. We will cover lists in a later section. For now, all you need to know is that this list holds a series strings, which must be comma-separated. For example, if you wanted to add a new item to this list, you could do the following:

```
colours = ['#', '|', 'V']
```

This adds the element 'V' to the list - note the added comma. Lists can be as large or as small as you like. They can even be empty, but must always be enclosed by square brackets [ and ].

### 3.5.2 TASK TWO: TRANSFORMATIONS

Write a Python program called `transformations.py` that takes a point on the cartesian plane (given by x and y coordinates as shown on the next page), transforms it using a sequence of operations, and prints the result to the terminal.

The operations should be as follows, in this order:

1. Reflect the point over the y-axis.
2. Shift the point up by 2.
3. Shift the point left by 6.5.
4. Vertically stretch the point by a factor of y.

---

Then output the final result by printing the new x and y coordinates to the screen.

At each step, you should update the coordinates by assigning the new transformation to the appropriate identifier. For example if I wanted to shift the point right by 1, I would use the following expression:

```
x = x + 1
```

to add one to the x-coordinate and update the variable.

Here is a snippet of code to get you started:

```
# These two variables store x and y coordinates
x = 1
y = 1

# PERFORM TRANSFORMATIONS HERE
# Reflect the point over the y-axis

# Shift the point up by 2

# Shift the point left by 6.5

# Vertically stretch the point by a factor of y

# Print the result
print(x)
print(y)
```

If you need a refresher on transformations, see [this quick reference sheet](#). The only additional knowledge you may need is that a vertical stretch affects only the y-coordinate and a horizontal stretch affects only the x-coordinate.

Sample inputs and outputs:

- $x = 1, y = 1$  should produce  $x = -7.5, y = 9$
- $x = 0, y = 5$  should produce  $x = -6.5, y = 49$
- $x = -5, y = 0$  should produce  $x = -1.5, y = 4$

## Transformation: Challenge Exercises

Once you have completed the transformation exercise, here are some additional challenges to try:

- Try to condense your code:
  - Combine the transformations to compute each coordinate (x and y) on only one line.
  - **Hard.** As an additional challenge, try to write the entire program, including the print statements, in one line. The declarations of x and y do not count for the purpose of this exercise.

---

- Pretty printing:

- Print the x and y coordinates on a single line, separated by a space. **Hint:** It may help you to recall that the `print()` function can take multiple arguments separated by commas.

- Print the x and y coordinates in the following format:

```
The coordinates of the new point are x y
```

where x and y are replaced by the coordinates.

- **Hard.** Finally, try printing them like this:

```
The transformed point is (x, y).
```

Look into [this page on string formatting](#) for some help.

## SECTION 4

---

### LAB 2 - USING THE TERMINAL

---

*This lab is designed to be used as a reference throughout CMPUT 274/275.*

*Take the time to complete and understand Sections 1-4 of this lab as soon as possible. The information in Section 5 is designed to challenge you once you feel comfortable with the basics of the terminal, so you can save it for later if you are new to Linux.*

*As you read each section and learn new skills, keep a terminal open and test each command before continuing to make sure you understand it, and complete all practice exercises in the Your Task sections. The only way to get truly comfortable with the terminal is to practice!*

#### 4.1 INTRODUCTION

For students who do not have previous experience with the Ubuntu (UNIX) operating system, getting used to the terminal environment can seem daunting. This section is designed to get you familiar with some common commands and functionality in the terminal. The information here will be useful to you in CMPUT 274 and beyond.

**Note:** Our discussion will focus specifically on the Bash shell and the terminal in the Ubuntu operating system used in the VM. The material covered here may not apply to other operating systems.

#### 4.2 TERMINOLOGY: TERMINAL, SHELL, COMMAND PROMPT

It may help you to learn a bit of terminology, so that you can use the proper keywords when you research the practical commands you will be using.

A **terminal**, in simplest terms, is an interface through which you can interact with the shell to type commands, communicate with your operating system, and run programs.

When you launch a terminal it will always run some program inside it. That program will generally by default be your **shell**. Using the Ubuntu OS, the default shell is [GNU Bash](#) or more commonly just **Bash**. The shell takes hand-typed commands and tells the operating system to execute them. It can also take Bash scripts and interpret logic within them, in addition to instructing the operating system to execute the commands contained in the script.

The shell is simply a program that runs other programs, and it is called a shell because it wraps around the kernel (your operating system).



---

### 4.2.1 THE COMMAND PROMPT

When you open the terminal and launch the Bash shell, you will see a **command prompt**. This is an invitation to type a command, and by default in Ubuntu, the prompt ends with a \$ for all users except **root**, which ends with a #.

## 4.3 ESSENTIAL LINUX BASICS

**Note:** If you are using a Mac, replace all instances of CTRL in all of the following sections with COMMAND.

### 4.3.1 WHAT IS A PATH?

A [path](#) is the location of a directory or file in a file system. There are two types of paths: **absolute** and **relative**. Whenever you refer to a file or directory, you are using one of these two types. Absolute paths specify a location (file or directory) in relation to the **root directory** of your machine. If the path provided begins with a slash character /, then the path is an absolute path starting at the root directory. Relative paths specify a location (file or directory) in relation to where we currently are in the system. They will not begin with a slash.

If the path provided begins with ~/ (tilda slash), then the path is starting at your home directory. If the path just begins with the name of a folder or file, with no slash or tilda slash, then the path is a relative path starting in your **current directory**. Another way of writing the current directory is . (dot) or ./ (dot slash). This is helpful for you to know when you want to specify files using the commands in the following sections.

[This page on basic navigation](#) contains more information about paths if you are interested.

### 4.3.2 YOUR TASK: PATHS

1. Open a new terminal and type the following command into the prompt:

```
pwd
```

This will print the absolute path to the directory where you are currently located. Notice that it begins with a '/'.

2. Move into the `arduino-ua` folder in your home directory by typing

```
cd arduino-ua
```

Now type `pwd` again. You should see that `arduino-ua` has been added to your path.

3. Open the file manager by clicking on the icon labelled Files on the left hand side of the screen. Using the graphical interface, Find the `arduino-ua` folder and move into it. Then click the following folders in order: `examples`, `arduino`, and then `BlinkCPP`. Right click anywhere in the folder and click “Open in Terminal”. This will open a terminal in your `BlinkCPP` folder.

(a) First type:

---

```
ls
```

To list the contents of the directory. This list should match what you can see in the GUI (graphical user interface).

- (b) Now type **pwd** again to see your new absolute path. Observe that the path you can see listed is the same as the path you took through the GUI.

- (c) Type

```
cd ../
```

To move into the previous directory.

- (d) Finally, type

```
cd
```

with no other arguments to return to your home directory.

### 4.3.3 SHORTCUTS

The terminal is full of shortcuts to help make your life easier. Here are a few of the most important ones to know:

1. When you enter commands, they are actually stored in a history. You can traverse this history using the **up and down arrow keys**. There is no need to re-type commands you have previously entered! You can usually just hit the up arrow a few times instead. You can also edit these commands using the left and right arrow keys to move the cursor where you want.
2. Use **CTRL+r to Search for a Command**: Using the up and down arrow keys can take a long time, especially if you've typed many commands since the one you want to find. If you press **CTRL+r** in the terminal, you'll see a message “(reverse-i-search)” with a prompt. You can then type the first letter(s) of a command you've issued before to find it, and then hit ENTER to run it. If you have a long Bash history, there might be many matching commands. Once you've typed some matching characters, you can press **CTRL+r** again to cycle through all similar commands.
3. You can use a handy little mechanism called **TAB completion** to avoid typos when typing the name of a file or directory. Whenever you start typing out a path, you can press the TAB key to invoke an action which will attempt to automatically complete the path. If nothing happens when you press TAB, there may be multiple options that start with the phrase you have typed. If this happens, you can press TAB again to see more possibilities.
4. You can **chain multiple commands together in one line** using the **&&** (double ampersand) logical AND operator. This can be extremely useful if you have a set of two or more commands that must be run together. For example, rather than run two separate commands, you can compile and run a C++ program using:

```
$ g++ program.cpp && ./a.out
```

Or you can upload a program to the Arduino and open the Serial monitor in one line using:

```
$ make upload && serial-mon
```

Note that the second command only runs if the first command succeeds without any errors (so the Serial monitor won't open in the above command if you have compile errors). If you are curious to learn about other operators like this, as well as more detail on how `&&` works, read [10 Useful Chaining Operators in Linux with Practical Examples](#).

#### 4.3.4 YOUR TASK: SHORTCUTS

1. Open a new terminal and press the up and down arrow keys to see which commands are already in your history. If you completed the previous exercise on paths, those commands should now be in your history.
2. Type `cd ar` into the terminal and hit the TAB key to see it autocomplete to `cd arduino-ua`.
3. Type `cd Do` and hit TAB. Nothing should happen here, since both your **Downloads** and **Documents** folders start with **Do**. Hit TAB a second time to see both options.
4. Practice chaining some commands together.

(a) For example:

```
cd arduino-ua && cd examples && cd arduino && cd BlinkCPP
```

will bring you to the BlinkCPP folder in the same way as typing

```
cd arduino-ua/examples/arduino/BlinkCPP
```

(b) Return to your home directory using `cd`. Try this command:

```
cd Documents && subl example.py
```

These two commands will create a python file called `example.py` and open it using the Sublime Text editor.

5. Practice using **CTRL+r** to find commands you've typed before.
  - (a) Try searching for `cd D`. This should complete to the command you just ran in the previous step, `cd Documents && subl example.py`.
  - (b) Then type `py` to find the last python script you ran. Press **CTRL+r** to see other recent python scripts in reverse chronological order. Keep in mind you can only run these commands again if you are in the directory where the script is located (unless you used an absolute path to the `.py` file).

#### 4.3.5 COPY AND PASTE

In the terminal, you cannot copy and paste as you normally would using keyboard shortcuts. Using **CTRL+C** will not copy text in the terminal if you have highlighted it with the mouse. Instead, the

---

sequence **CTRL+C** is actually used to send a kill command to a process that is running. Similarly, **CTRL+V** will not paste text into the terminal.

Instead, you can use **CTRL+SHIFT+C** to copy from the terminal and **CTRL+SHIFT+V** to paste into it. This can be useful in many cases, including during morning problems when you wish to paste the sample input into the terminal. You may need to replace **CTRL** with **COMMAND** on Mac.

**Note:** Be careful when copying text or commands from a pdf! You are not copying plaintext so some characters may be different or the spacing may be incorrect. Always check that what you have copied looks correct before hitting enter.

#### 4.3.6 ENDING A PROCESS

Sometimes, you may end up with a program that will not terminate, either because it is waiting for an event that has not happened (such as input from the user), because it is using too many computer resources, or because it is stuck in an infinite loop.

You can use **CTRL+C** to send a kill command (specifically by sending the signal **SIGINT**, if you are curious) to a process that is currently running. Do not use **CTRL+Z** for this purpose! **CTRL+Z** only stops a process. It will not kill the process, but will only “pause” it and put it into the background. You can type:

```
ps
```

To see a list of currently running processes, with their process identifiers on the left. Each process has a unique ID.

Note that you can do many more things with processes that are currently running. For example, you can shift processes to run in the background, and send various signals to a running process. [Click here if you want to learn more](#) about how to do this.

#### 4.3.7 YOUR TASK: ENDING A PROCESS

1. Open a terminal in your home directory. Navigate to your Documents folder and open a new python file called **stuck.py**. *If you are having trouble with this, see the previous task section for commands that will help you.*
2. Type the following two lines of code into the file:

```
while True:  
    pass
```

This will create an **infinite loop** that will do nothing forever. You will learn more about infinite loops in Lab 4.

3. Type

```
ps
```

---

to see the processes which are currently running, before doing anything further. This gives you a baseline to see how things change.

4. Run the program using

```
python3 stuck.py
```

The process will run, but nothing will happen. Press **CTRL+C** to terminate the process.

5. Type **ps** again. You should not see any new processes.
6. Run the program again, and this time press **CTRL+Z**.
7. Type **ps** again. This time, you should see a new entry labelled as a python process. This is your infinite loop!
8. Terminate the background process by typing:

```
kill -SIGKILL 8842
```

where 8842 is replaced by the process ID shown to you by **ps**. Note that **SIGKILL** is the strongest termination signal to send to a process. Often, you can just type **kill 8842**, which sends the signal **SIGTERM** by default. This may be enough to end the process, but if it does not cooperate (ie, you still see it when you type **ps**), you may need the stronger signal **SIGKILL**.

9. Type **ps** a couple more times. The first time, you may see a message letting you know that the process was killed, but eventually the list should be the same as at the beginning.

### 4.3.8 STANDARD INPUT, OUTPUT, AND ERROR STREAMS

In Linux and other Unix-like systems, there are three default **streams** for input, output, and errors. Each stream has an associated **file descriptor number** used for identification:

- 0. standard input (stdin), which is the normal source of input for your program.
- 1. standard output (stdout), used for normal output from your program.
- 2. standard error (stderr), used for any error messages or diagnostics from your program.

Take a moment to read [this page](#) up to (but not including) the section on Stream Redirection. It is a good idea to understand where your input is coming from and what happens to the output your program produces.

## 4.4 USEFUL COMMANDS

### 4.4.1 PATHS, DIRECTORIES, AND NAVIGATION BASICS

- **ls (list): lists all the files/directories in your current directory.**

Usage: Type **ls** alone to see the list of objects. You can use **ls -a** to see hidden files (these typically start with a **.** character). There are also a number of other command-line arguments (additional options) which you can use with **ls**, [detailed here](#).

- 
- **cd (change directory):** changes your current directory to whatever path you provide.

Usage: Type `cd` and then the directory you wish to change to. For example, `cd ~/arduino-ua` will change the working directory to the directory `arduino-ua` in the home directory.

- using `cd ~/` or `cd` alone will bring you back to your home directory
- using `cd ..` or `cd ../` will move to the previous directory

- **pwd (print working directory):** this prints the full path (starting at `/`) of your current directory.

Usage: Simply type `pwd` to view the path. Note that your command line prompt also tells you the name of your current directory. In the picture below, you can see that the command line prompt has some machine details followed by a colon and the name of the current directory (followed by the user name and the dollar sign `$` which marks the command line prompt. Depending on your operating system, you may or may not see your user name).

#### 4.4.2 MAKING NEW FILES AND DIRECTORIES

- **subl (Sublime):** used to create a new file using the Sublime Text editor.

Usage: Type `subl` and then the name of the file you wish to create. The file will be opened using the editor and stored in the current directory. You can also open files using other editors (all of these examples are available on the VM):

- **vim <filename>:** Opens a new file in Vim.
- **atom <filename>:** Opens the new file in Atom.
- **gedit <filename>:** Open the new file in Gedit.

- **mkdir (make directory):** creates a new directory with the name/path provided.

Usage: If you just type `mkdir dirname` then it creates the directory in your current directory. If you use `mkdir ~/dirname` then it creates that directory in your home directory. You can specify any path as you normally would.

**Example:** `mkdir ~/exercise1`

This will create a new directory in the home directory called `exercise1`.

- **more:** allows you to view the contents of a file in the terminal

Usage: Type `more` and then the name of the file you wish to examine.

**Example:** `more example.txt` This will print the contents of `example.txt`, if it exists.

#### 4.4.3 MOVE, REMOVE, AND COPY

- **cp (copy):** copies a file to a new location.

Usage: `cp` requires two arguments. You must type `cp`, the path to the file you want to copy, and the location to copy the file to. For example `cp ~/filename ~/Documents` copies a file called `filename` from my home directory to `~/Documents`.

---

**Example:** `cp ~/arduino-ua/Makefile_Example/Makefile .`

This will copy the Makefile from the Makefile\_Example directory in arduino-ua to the directory you are currently in. Pay special attention to the second parameter! In this case, it is just “.”, meaning “your current directory”. If you want to copy to the directory where you are currently located, you need to specify “.” (note that typing the absolute path to your current directory also works).

- **mv (move):** moves a file from the specified old location to the new one.

Usage: Similar to copy above. For example, `mv ~/exercise1/sos.ino ~/backups` will move the file `sos.ino` into a directory called `backups` in the home directory. Note that unlike `cp`, this will remove the file from the old location before copying it to the new one.

- **rm (remove):** removes (deletes) a file. Note that this is permanent.

Example: `rm blink_backup.ino`

This will delete the file called `blink_backup.ino` in the current directory. In our VM, you will get a prompt asking you if you really want to remove it. On other systems, this safety feature may not be present, so be careful!

- **rmdir (remove directory):** remove an empty directory. This is also permanent.

Usage: Type `rmdir` and the name of the empty directory to remove. This will only succeed if the directory is empty (i.e. has no other files or subdirectories). If the directory is not empty, you will need to remove all files from inside of it first. One way to do this is to type `rm -r directory_to_remove`. This uses the option `-r` to **recursively** remove the directory AND everything inside of it (including any subdirectories).

#### 4.4.4 COMPRESSING AND EXTRACTING FOLDERS

You will be required to submit many assignments as `.tar.gz` or `.zip` archives, so it is useful to know how to create them from the command line. The commands used are either “tar” or “zip”, followed by a number of command-line arguments. The tar command is generally preferred over zip. **Note that it is very important to ensure that you type all the arguments in the correct order.** Doing otherwise has the potential to overwrite your file. Be careful!

- **Creating .tar.gz files:** To create a `.tar.gz` archive, use the following command:

```
tar -zcvf tar-archive-name.tar.gz source-folder-name
```

where `tar-archive-name` is the name of the archive you wish to create and `source-folder-name` is the name of the original folder you wish to compress.

- **Extracting .tar.gz files** To extract a compressed `tar.gz` archive, use the following command:

```
tar -zxvf tar-archive-name.tar.gz
```

where `tar-archive-name` is the name of the archive you wish to extract. Note that the `c` flag above is now changed to an `x`.

- **Creating .zip files:** To create a `.zip` archive of a folder, use the command

---

```
zip -r zip-archive-name.zip source-folder-name
```

where `zip-archive-name` is the name of the archive you wish to create and `source-folder-name` is the name of the original folder you wish to compress.

- **Extracting .zip files** To extract a compressed zip archive, use the following command:

```
unzip zip-archive-name.zip
```

where `zip-archive-name` is the name of the archive you wish to extract.

Note that you it is possible to compress any number of files, and not just one folder. If you need to do this, it is up to you to research the exact command you will need.

#### 4.4.5 YOUR TASK: WORKING WITH TERMINAL COMMANDS

**Part 1: Walkthrough.** This first section will give you instructions and then the exact command needed to complete the task.

1. Open a terminal to your home directory. Navigate to your Documents folder and create a new folder called `first_folder`.

```
cd Documents && mkdir first_folder
```

2. Move into `first_folder`. Check the contents to confirm that it is empty

```
cd first_folder
ls
```

3. Use Sublime Text to create a text file called "hello.txt".

```
subl hello.txt
```

4. Type some text into the file (whatever you like) and then save and close the file. Now switch back to the terminal and type:

```
more hello.txt
```

To see the exact text you just typed.

5. Now create a second file called "helloworld.py". Type some quick Python code (I suggest `print("Hello World!")`) and then save and close the file.

```
subl helloworld.py
```

6. Move back out of the `first_folder` directory into your Documents, and create a second folder called `second_folder`.

```
cd ../ && mkdir second_folder
```

7. Now, move the file "hello.txt" from `first_folder` to `second_folder`.



---

```
mv first_folder/hello.txt second_folder
```

8. Move into `first_folder` and use `ls` to check that the file was moved.

```
cd first_folder
ls
```

Note that this would also work:

```
ls first_folder
```

9. Now repeat the previous step for `second_folder` to check its contents as well.

```
cd ../second_folder
ls
```

**Part 2: On Your Own.** This section will give you instructions without listing any specific details about how you should complete the tasks. However, you will always be able to do so using only the commands taught in this section. This section assumes that you have completed the previous set of problems successfully and have the folders `first_folder` and `second_folder` in your Documents folder. If you have not, take the time to do so now.

1. Open a terminal and navigate to your Documents.
2. Use `tar` to compress the `first_folder` folder into a `.tar.gz` compressed archive called `first_folder.tar.gz`.
3. Type `ls` to check the contents of the Documents directory. Note that the archive was created and that the uncompressed folder `first_folder` still exists. Let's get rid of the uncompressed folder:
  - The `rmdir` command won't work on a non-empty folder (note that you could use `rm -r`, but for the sake of practice we will not do that here). Instead, move into the `first_folder` folder.
  - Empty the contents of `first_folder`. Hint: The only file inside of it should be "helloworld.py". Check that the folder is empty before you proceed.
  - Now move back into Documents.
  - Delete the `first_folder` folder.
4. Now let's see what's in our compressed archive. Extract it.
5. Check the contents of the folder `first_folder` using `ls`.
6. Delete the `first_folder.tar.gz` compressed archive.
7. Copy the "hello.txt" file from `second_folder` back into `first_folder`. There should now be a copy of this file in both folders. Use `ls` to confirm this.

---

#### 4.4.6 REDIRECTING INPUT AND OUTPUT

Typically, your program will take input from `stdin` and output to `stdout`. For example, when you use `input` and `print` in Python or `cin` and `cout` in C++, you are working with these standard streams.

However, it is possible to **redirect** the input and output of a program using the Bash terminal. This allows you to easily read input from and write output to files other than `stdin` and `stdout`.

1. The ‘<’ symbol is used for **input redirection**. Instead of reading from standard input, you can choose to read from a file instead.

**Example.** You are working on solving a morning problem called “Rock Paper Scissors” and need to test your code using a specific test case. However, to do this, you have been wasting your precious minutes copying and pasting the same test case over and over again every time you run the program. Instead, you can save the test case in a file, in this case called “testcase”, and get the input from the file using the redirection operator:

```
$ python3 rockpaperscissors.py < testcase
```

Now every time you want to run the testcase stored in the file, you can simply use this command.

2. The ‘>’ symbol is used for **output redirection**. Be careful with ‘>’, as it will overwrite the contents of the given file!

**Example.** Using the above example, you now want to read from the “testcase” file and then print to another file called “out”. You can combine the commands like this:

```
$ python3 rockpaperscissors.py < testcase > out
```

The above command creates a file named “out” and writes to it the output of your program.

#### 4.4.7 YOUR TASK: MORNING PROBLEM DRY RUN USING TERMINAL COMMANDS

Do this section once you have completed Lab 4: Morning Problem Preparation. It is very important to know the terminal environment well when solving morning problems, as you will have a limited amount of time to complete them. Go through these steps to practice using the terminal to solve a morning problem.

1. Go to eClass and download the compressed archive (.tar.gz) of the problem you are planning to solve. You can find the problems in the Morning Problem tab. Save the archive in the folder of your choice.
2. Open a terminal and navigate to the folder containing the compressed archive.
3. Extract the archive using `tar`.
4. Move into the new folder (which will have the same name as the archive). Use `ls` to check the contents of the directory. You should see, in some order:

```
soln    testcases    testcenter.ini    opentestcenter.sh
```

5. Move into the `soln` folder.

- 
6. Open the solution file using Sublime Text. The solution file will either be a .cpp or .py file, depending on whether you are solving the problem in C++ or Python.
  7. Code your solution to the problem, and then make sure the solution is saved.
  8. Now we will create a file to store a testcase. Open a new file in the `soln` folder called “testcase.txt”. Copy and paste a testcase from the pdf description into “testcase.txt”. Make sure there is an extra blank line at the end - that is equivalent to the user pressing ENTER at the end of the input.
  9. Return to the terminal and run the solution file, using redirection to take input from “testcase.txt” rather than standard input (see the above section for help doing that).
  10. Once your code produces the expected output, use `cd` to return to the main problem directory. There, use

```
./opentestcenter.sh
```

To open and run the Test Center. Run all test cases and celebrate when everything passes successfully! Upload the solution file to eClass.

## 4.5 WAYS TO LEARN MORE

There are many, many more things to learn about the Bash shell that can drastically improve your development experience. This introductory lab barely scratches the surface of the many things you can do in the Bash terminal. Some of these extremely useful skills include (but are not limited to):

- Permissions, `chmod`, and the `sudo` command.
- How to install packages yourself using `sudo apt-get`.
- How to use pipes for advanced input and output redirection.
- [Hidden files and directories](#).
- Filters, used to accept and transform textual data in some way (including `find` and `grep`).
- Regular expressions, used to search for patterns in text.
- How to create your own Bash scripts.
- Process management to view and control running processes.

### 4.5.1 BASH TUTORIALS

If you want to learn more than is contained in this lab about the commands and functionality available to you in the terminal, there are many useful resources available to you on the internet. Here is a quick list of recommended Bash tutorials you can use to learn more.

**General Tutorials:**

- 
- [Ryan's Tutorials: Linux Tutorial](#): An essential 13 part tutorial that will get you used to working with many of the tools of Linux. Do this! Some of these pages have already been referenced in this lab. The most important pages to read for CMPUT 274/5 are Sections 1-5 and 11. The [cheat sheet](#) found here is also very useful.
  - [UNIX Tutorial For Beginners](#): Another tutorial recommended by previous CMPUT 274/5 instructors, with emphasis on Tutorials One, Two, and Four.

### A Few Specific Topics to Learn About:

- **Users and Permissions:** The `sudo` command used to gain administrative privileges. Note that on the VM, the password to gain administrative privileges (eg, for installing packages) is the same as the password to enter the VM: `cmput274`.
  - [Linux Commands for Beginners: Sudo](#).
  - [A quick summary explaining the difference between su, sudo bash, and sudo sh](#).
- **Bash Shortcuts for Maximum Productivity:** A short, quick-reference summary of the rich array of convenient keyboard shortcuts that the Bash shell supports.
- **Installing Packages Using `sudo apt-get`:** There is a free, simple package manager installed by default on Ubuntu operating systems. This is your main mechanism for installing new packages or applications, so it is very important to know how it works.
  - [How To Manage Packages In Ubuntu and Debian With Apt-Get and Apt-Cache](#).
  - [25 Useful Basic Commands of APT-GET and APT-CACHE for Package Management](#).
- **Grep and Regular Expressions:** `grep` is a powerful tool that can be used to search a text for words or phrases matching a given pattern. To specify a search pattern, you use regular expressions (or regex).
  - [Grep Manual](#): The official documentation for `grep`, including instructions on regular expressions.
  - [Linux grep command usage with examples](#): Includes instructions for installation if you do not already have it installed by default.
  - [Grep Command in UNIX/Linux](#): Another good tutorial to start learning.
  - [Ryan's Tutorials: Regular Expresions Tutorial](#): An excellent resource for learning regular expressions from beginner to advanced.
- **Writing Your Own Bash Scripts:** You can write your own scripts containing Bash commands that can be executed quickly and easily in order using only one command. One example you may be familiar with is `opentestcenter.sh`, a Bash script used to open the Test Center during morning problems.
  - [Ryan's Tutorials: Bash Scripting Tutorial](#).
  - [Bash Scripting Tutorial for Beginners](#) Another good tutorial to get you started.
  - [How to Create a First Shell Script](#): A resource that contains some good basic information as well as a few exercises you can use to practice on your own.

- 
- **More on Input Redirection and Pipes:** The section on input and output redirection above taught only the basics. You can use a tool called a pipe (the character '|') to direct the output of one program to become the input of another. A few advanced tips:
    - To add content to a file rather than overwriting it, use the '>>' output redirection operator.
    - You can use the [tee command](#) to send output to multiple streams at once, effectively duplicating the output.

Here are a few additional resources and tutorials if you are curious to learn more:

- [An Introduction to Linux I/O Redirection](#). This page was linked to above and is an excellent resource. It does not assume any prior knowledge and makes sure to explain concepts thoroughly.
- [Input Output Redirection in Linux/Unix Examples](#). This page gives a quick overview and also discusses redirecting error messages, which print to their own stream called standard error (stderr).
- [Bash One-Liners Explained, Part III: All about redirections](#). A bit more advanced, this page gives excellent background on how the streams themselves work as file descriptors and how to use custom file descriptors.

#### 4.5.2 MAN PAGES

The Bash shell comes with a builtin tool called [man pages](#). These manual pages are a set that describe and explain every command available on your system. They contain useful information about what each command does, the specifics of how to run them, and what additional command-line arguments they can accept. Sometimes, they may seem difficult to navigate or understand, but like any documentation, they can provide extremely useful information.

To access the man pages, type

```
man <command>
```

where <command> is the command you wish to know about. For example, try typing `man ls` into the command prompt to see what happens. You can even type `man man` to read the manual page about the manual itself!

Here are a few resources if you are interested:

- [Ryan's Tutorials: Manual Pages! Your reference on Linux](#).
- [MAN PAGES: A tutorial](#).

## SECTION 5

---

### LAB 3 - DICE SIMULATOR WITH OUTCOMES

---

*You are with a group of friends and want to determine where you should go for dinner. Everyone has ideas, but no one is actually making the decision.*

*You are an avid Dungeons and Dragons player, and as the Dungeon Master, you want to quickly determine the outcome of a roll for one of your players.*

*You want to surprise your dad with a father's day gift. You have a ton of possibilities, but can't decide which to make. Why not let chance decide?*

In this section you will make a simple dice roller that will simulate a six-sided die and assign an outcome to each roll. When the program is run, a random number from 1 to 6 will generate and the outcome associated with that number will be printed to the screen. For example, the outcome attached to the number 1 could be “Make dinner at home”. Then, if the program rolls a 1, the string “Make dinner at home” is printed to the screen.

#### Learning Outcomes:

- Know how to import a module in Python, and how to use the `random` module and the `randint` function.
- Understand conditional statements (if/elif/else) and indentation in Python.
- Know the basics of lists and list indexing.
- Understand the importance of code quality.

#### 5.1 THE RANDOM MODULE

To make our dice simulator, we will need a way of generating pseudo-random numbers. We could do this by ourselves, but it would be tedious and complicated. Fortunately, someone else has already written this code. All we need to do is **import** the module containing the code so that we can use it.

Using the simplest definition, a **module** is any file containing Python code. Usually, a module contains **functions** that can be called using zero or more **arguments**. Here, we will import a module that contains functions for generating pseudo-random numbers: `random`.

By importing the `random` module, we will have access to all the functions it contains. This includes the function we will need to use: `randint`, which generates a pseudo-random integer in a given range.

---

There are many other functions in the `random` module which we will not use in this exercise. See the [documentation](#) of the `random` module to learn about the others.

Resources to learn more:

- [A tutorial on Python 3 modules.](#)
- [Python Documentation: Modules.](#)

### 5.1.1 IMPORT STATEMENTS

#### Method 1:

You can import a module using the following syntax:

```
import module_name
```

where `module_name` is the name of the file you wish to import.

If you use this syntax and you wish to access a function called `function_name` from within the module, you must call it like this:

```
module_name.function_name()
```

#### Method 2:

If you only need one or two functions (or methods) from the module, you can use this syntax instead:

```
from module_name import function_name
```

Using this syntax, if you wish to call `function_name`, you can simply use:

```
function_name()
```

Because we only need the `randint()` function for our dice simulator, we will use this second method.

## 5.2 PRACTICE IN THE INTERPRETER

I just told you about a new Python function, `randint`. Now that you know how to import it, you need to find out how it works. There are two steps to this process:

1. Read the documentation for the `randint` function. You will find it [on this page](#) under section 9.6.2 (Functions for integers).
2. Practice calling the function in the Python interpreter. Experiment with different invocations to see exactly how it works.

---

After finishing step 1, open a terminal and launch the Python 3 interpreter. Even in the interpreter, you need to import the `randint` function before you can use it, so type the following into the interpreter prompt:

```
from random import randint
```

You should not see any output from this line.

### 5.2.1 CALLING THE RANDINT FUNCTION

On the next line, let's try **invoking** the `randint` function. The `randint` function is **defined** somewhere in the `random` module, but it does not get **executed** until it is called.

From reading the documentation, you should know that `randint` takes as input two [parameters](#), `a` and `b`, separated by a single comma.

Try typing each of the following lines into the interpreter. You should see an error message after each one, because you are not invoking the function correctly:

```
>>> randint()  
>>> randint(3)  
>>> randint(3,2)  
>>> randint(4.3, 2.4)
```

**Note:** To call a function, you must use round brackets! Using square brackets (`[]`) or curly brackets (`{}`) is incorrect and both will return an error.

Square brackets are typically used to **index into** lists or other containers (you will see this later). If you do this, the Python interpreter will return a `TypeError` because you cannot index into a function (or in this case, a method):

```
>>> randint[1,2]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: 'method' object is not subscriptable
```

Using curly brackets will return a `SyntaxError` simply showing that the **structure** of the line is incorrect:

```
>>> randint{1,2}  
File "<stdin>", line 1  
    randint{1,2}  
           ^  
SyntaxError: invalid syntax
```

By contrast, all of these are correct invocations of the `randint` function:

```
>>> randint(0,0)  
>>> randint(1,2)  
>>> randint(-5, -1)  
>>> randint(1, 100)
```



---

Try this out for yourself by testing different ranges and generating some pseudo-random numbers. Keep at it until you feel comfortable with the `randint` function. Notice that both  $a$  and  $b$  are included in the range! So `randint(1, 100)` generates a number between 1 and 100 inclusive.

**From now on, whenever you learn a new function in Python, repeat this process.** Test the function to learn exactly how it works! This will also help you better understand what the documentation means.

## 5.3 VERSION 1: SETTING UP THE DICE ROLLER

Our first goal is to get a simple version of the dice roller running. Once we have random number generation working, we can add our outcomes.

Open a new Python program in Sublime Text and call it `dice_roller.py`. First we will add the import statement needed to get the `randint` function:

```
from random import randint
```

Notice that there are no round brackets after the word `randint`. Those are only needed when we **call** the function.

Below that line, we want to call the `randint` function to generate a number between 1 and 6 (inclusive) and store the pseudo-random number it generates into a variable. Add the following lines to your program, save the code, and run it from the terminal.

```
# generate the pseudo-random number
number = randint(1,6)

# display the number
print(number)
```

Every time you run your program with these three lines, you should see a different number between 1 and 6.

## 5.4 CONDITIONAL STATEMENTS (IF/ELIF/ELSE)

We want to improve our dice roller so that it will output a different string depending on the number generated. To do this, we need to create code that will run only if a certain **condition** is true.

To do this, we need to introduce the concept of a **conditional statement**. You can [read this very good tutorial on conditional statements](#) for a more thorough explanation.

### 5.4.1 CONDITIONS

A **condition** is an expression that evaluates to either True or False. A conditional statement allows your program to make a decision: evaluate some code if the condition is true, and do not evaluate it otherwise.

Often, this is done by **comparing** two values in some way. Below is a table of comparison operators

---

that can be used to create conditions in Python.

Table 5.1: Comparison Operators in Python 3

operator	function
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal

Open the interpreter and type the following expressions. All of them are conditions, and you should see either **True** or **False** echoed back in the interpreter. Try to predict what each will do before you see it evaluated.

```
5 < 4
12 == 13
12 == 12
12 != 13
'b' > 'a'
2 >= 1
'hello' >= 'hello'
```

**Note:** Be careful that you do not confuse the `=` assignment operator and the `==` comparison operator! This is a common mistake made by beginners and experienced programmers alike. The `=` operator is used to assign a value to an identifier, while the `==` checks to see if two values are equal. They are not interchangeable!

#### 5.4.2 BOOLEAN OPERATORS: AND, OR, & NOT

It is also possible to chain conditions together using the built-in Python keywords **and**, **or**, and **not**. These more complex expressions still return a single **boolean value** when fully evaluated: either **True** or **False**. They work as follows:

- An expression **X and Y** returns **True** only if both **X** is true and **Y** is true.
- An expression **X or Y** returns **True** if at least one of **X** and **Y** is true.
- The expression **not X** returns **True** if **X** is false, and **False** if **X** is true.

Here are the [truth tables](#) describing the results of combining the expressions **a** and **b** using boolean operators in Python ([image taken from here](#)):

---

a	not a	a	b	a and b	a or b
False	True	False	False	False	False
True	False	False	True	False	True
		True	False	False	True
		True	True	True	True

*Truth-table definitions of bool operations*

### Learn More:

- Read [Boolean Operators: A Cheat Sheet](#).
- Complete the [Computing Science Circles tutorial](#) on else, and, or, and not and start reading at the section called Boolean Operators.

## 5.5 IF STATEMENTS

The syntax for a basic if statement looks like this:

```
if <conditional expression>:
    <block of code to evaluate if the condition is true>
```

### 5.5.1 INDENTATION IN PYTHON

Notice that each line in the [block of code](#) under the if statement must be **indented** in order for the if statement to be evaluated correctly. Python programs are **structured using indentation**. To indicate a block of code in Python, you must indent each line of the block by the same amount. The two blocks of code in our example if-statement are both indented four spaces, which is a typical amount of indentation for Python.

If you do not have proper indentation in your program, you will see an **indentation error**.

More about indentation in Python:

1. [Flow of Control in Python: Code Blocks and Indentation](#)
2. [Structuring with Indentation](#)

Another important thing to notice is the [colon](#), :, at the end of the conditional expression. The colon indicates to the Python interpreter that the conditional expression has ended, and that the block of code under the if statement is beginning.

### 5.5.2 ELIF AND ELSE STATEMENTS

In a program, we may need to do something even when an if statement evaluates to false. For example, our dice roller needs to output one string when the integer generated is 1, and do something else when the integer generated is not 1. For this, we need an else statement.

---

An else statement is optional, but it must always follow an if statement. There can be at most one else statement for every if.

Here is the syntax for an else statement:

```
if <conditional expression>:
    <block of code to evaluate if the condition is true>
else:
    <block of code to evaluate>
```

The block of code under the else statement is only evaluated if the conditional expression is false.

However, our dice roller needs to make more than two decisions. We need to introduce a new concept for this, the `elif` statement.

The syntax for an elif statement looks like this:

Here is the syntax for an else statement:

```
if <first conditional expression>:
    <block of code to evaluate if the first condition is true>
elif <second conditional expression>:
    <block of code to evaluate if the second condition is true>
else:
    <block of code to evaluate>
```

You can have zero or more `elif` statements. No matter how many `elif` statements there are, you must have an `if` and an `else` to contain them.

### 5.5.3 YOUR TASK: PRACTICE WITH CONDITIONALS

This section contains some optional exercises which provide some simple practice with conditionals. The first has a solution, but the others are up to you.

In Sublime, open a new Python file. Use a file to write these exercises rather than the terminal.

```
subl conditionals.py
```

For now, don't worry about reading input from the user. In this exercise, you can simply hard-code the values of each variable. For each exercise, you should make sure to test that all cases are correct by setting the variable to different values!

Exercises:

1. Write a program that determines if a number  $x$  is positive, negative, or zero. The program should print "Positive", "Negative", or "Zero" accordingly. A sample solution to this problem can be found at the end of the section. Try to implement it yourself before looking at the solution!
2. Write a program to determine what kind of gear you will need to go outside depending on a temperature,  $t$ . For the purposes of this question, we will assume that it is always sunny. If the temperature is:

- greater than -40 and less than or equal to 0, print “Winter coat”
- greater than 0 but less than or equal to 10, print “Light jacket”
- greater than 10 but less than or equal to 20, print “Sweater”
- greater than 20 but less than or equal to 40, print “Sunhat”
- If the temperature is less than or equal to -40 or more than 40 degrees Celsius, print “Not going outside!”

If you disagree with these thresholds, feel free to change them as you see fit.

3. Write a program that takes two strings, *a* and *b*, and determines which are longer. **There is a builtin function called `len` that may help you.** The documentation of the `len` function can be [found here](#).
  - Print the output to the terminal in the format “L is longer than S” where L is the longer string and S is the shorter string. For example: “greetings is longer than hello”
  - If the strings are the same length, print “The strings are the same length”

To gain some practice with the `randint` function, modify exercises 1 and 2 to generate a random integer in a valid range to test your program. Make sure you print the integer you generate to the screen or you will have no way to know whether your program outputted the right answer!

Here is the solution to the first exercise:

```
# change this value of x to test the program
x = 4

# make sure to check all three possible conditions!
if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

## 5.6 VERSION 2: ADDING OUTCOMES TO THE DICE ROLLER

In this section, we will add the outcomes to our dice roller. You will finish this process and create Version 2 in the *Your Tasks* section below.

Here is the basic code for the dice roller we developed in an earlier section.

```
from random import randint

# generate the pseudo-random number
number = randint(1,6)

# display the number
print(number)
```

---

Let's add a variable for each outcome. In the example, the outcomes are empty. You will fill the outcome variables with strings of your choice and add the conditional statements in the next section.

```
from random import randint

# here are the six outcomes
outcome_1 = ""
outcome_2 = ""
outcome_3 = ""
outcome_4 = ""
outcome_5 = ""
outcome_6 = ""

# generate the pseudo-random number
number = randint(1,6)

# change this to print the outcome string instead of the number
# you will need to add conditional statements here
# eg, if number == 1...
print(number)
```

## 5.7 YOUR TASK: FINISH VERSION 2

Your task is to finish Version 2 of the dice roller code (started in the previous section).

Finish this code by

1. Filling in the six outcome strings.
2. Adding conditional statements such that `outcome_1` is printed when a 1 is rolled, `outcome_2` is printed when a 2 is rolled, and so on.
3. Test your code. Instead of randomly generating the number, set the variable `number` to each possible outcome to make sure they all work correctly.

## 5.8 ISSUES WITH VERSION 2 OF THE DICE ROLLER

Although Version 2 is functional and has all the required features, the quality of the code is low. Why?

1. **Rigidity:** What if we wanted to use a 12-sided die rather than having only six outcomes? We would have to double the size of our program, which takes time and many lines of code. Our program is not easily changeable.
2. **Length:** The program is much longer than it needs to be. We use a different conditional statement for each outcome! We can make it much shorter.

What we really want is to condense or get rid of all the conditional statements, and simply access the  $i$ th outcome directly (where  $i$  is the number rolled by the dice roller). For example, we want to access the 4th outcome when the number 4 is rolled.

---

Fortunately, Python has a convenient [data structure](#) we can use to do this quickly and easily. Data structures allow programmers to organize information in a more convenient way, to make programs shorter, easier to write, or to run more quickly.

## 5.9 LISTS

One of the most basic data structures in Python is a **list**. A list is a **container** that can be used to hold any number of objects of any type. For example, in Python you can create a list of integers:

```
my_list = [3, 9, 27, 9, -2, 0]
```

Or floats:

```
my_list = [0.5, -2.5, 4.6, 7.3, 2.0]
```

Or anything you like:

```
my_list = ["Hello!", [1,2,3], 4, 7.3, "Five"]
```

When you declare a list, use the following syntax:

```
<identifier> = [<comma-separated objects>]
```

Lists can store a collection of related objects in a way that makes them easy to access and manipulate. For example, imagine that you have a list of animal strings that you want to include in your program: “elephant”, “dog”, “cat”, “rat” and “mouse”. You want the animals to be ordered by size with the largest one first. Without a data structure like a list you would need a different variable for each word:

```
word0 = "elephant"  
word1 = "dog"  
word2 = "cat"  
word3 = "rat"  
word4 = "mouse"
```

This is extremely cumbersome. Instead, let’s use a list:

```
word_list = ["elephant", "dog", "cat", "rat", "mouse"]
```

A list is a **sequence type**, which means that a list is an **ordered** collection of objects. Each element in a list is called an **item**. Each item in the list has an **index** which corresponds to its position in the list, **starting from zero**. This lets us keep track of the order of the items in our animal list because we know the index of each animal.

### 5.9.1 ACCESSING LIST ELEMENTS

Given our animal list above, how do we get the particular word we want? We need to **index into** the list in order to use or modify a particular element.

Each element in a list has a position, or **index**. Indexing starts from zero in Python (and in most programming languages). In our `word_list` the string “elephant” is element 0, “dog” is element 1, and so on.

---

We use **square brackets** to access an element of a collection. For example, to access the 3rd element of a list called `my_list`, do the following:

```
my_list[2]
```

We will learn more about lists later, as there are many more things you can do with them. For now, this information is enough to solve the problem.

More information on lists:

1. [Python 3: Lists](#)
2. [Understanding Lists in Python 3](#)

## 5.10 YOUR TASK: CREATE VERSION 3

You will improve the code from Version 2 in several ways. Some of the changes will improve code quality, and the challenge exercise will make your dice roller more versatile.

Your tasks are as follows:

1. Change the outcome variables so that they are all stored, in order, in a list. You can call this list `outcomes`.
2. Change the call to `randint` so it generates an index from 0 to 5 instead of a roll from 1 to 6.
3. Replace your conditional statements and print the correct outcome by indexing into the list using the randomly generated index.

### Challenge Exercise

[Hardcoding](#), or the practice writing literal values directly into your code, makes the code rigid and inflexible. We have removed a lot of hardcoding in our program by removing the outcome variables and conditional statements, but there is one final section of the program that is hard-coded: the index of the final position in the list. Right now, that is hardcoded as the number 5.

**Your task:** Find out how to change the call to `randint` so that it is dependent on the length of the list. This would allow you to modify your `outcomes` list to have any number of items (greater than 1) and have your program still work successfully.

**Hint:** Think: What is the final position in any list? For example, in this list, it is 4:

```
my_list = [0.5, -2.5, 4.6, 7.3, 2.0]
```

Note: You may want to look into the [len function](#).

## 5.11 WHY ALL THESE VERSIONS?

To an experienced programmer, making three different versions of a simple dice roller probably seems silly. However, making three versions allowed us to do several things:



- 
1. **Teach several important Python skills to beginners incrementally.** Each new Python feature was introduced when it was needed by the program we wanted to create.
  2. **Practice evaluating code, spotting flaws, and then rewriting the code to make it shorter, less rigid, and more correct.** You will almost never write perfect code on your first try. Although versioning may not be so important for a simple program like this, finding ways to make quality improvements to your code can be extremely important for large programs.
  3. **Get a working version of the code complete before trying to implement advanced features.** Attempting to do everything at once often causes bugs that are difficult to track down. It is hard to find the source of a problem when your code base is huge and the problem could be anywhere. Put pieces of your code together one at a time and test each step. Doing this will let you localize bugs that will be nearly impossible to track down later!

It is also important to acknowledge that this version of the dice roller is by no means perfect. or one thing, `randint` function is a [pseudorandom number generator](#), so the numbers it produces are not truly random. For another, you have to rerun the program for every single roll, which is tedious when you want to generate many numbers. Finally, if we wanted to change the outcomes list we would have to modify our Python code directly, which makes this program much more difficult for a non-programmer to use.

## SECTION 6

---

### LAB 4 - PREPARING FOR MORNING PROBLEMS: INPUT AND BUILT-IN PYTHON FUNCTIONS

---

Early in CMPUT 274, you will be expected to solve a programming problem during the first 30 minutes of every class. These challenging problems will ask you to use your problem solving and programming skills to code a solution. Each of these problems will expect you to take **keyboard input** from the user, calculate a solution to the problem, and **output** the correct result.

For morning problems, understanding the intricacies of reading input in Python will be very important. For most of the problems, you will be expected to read input quickly and easily on your own. If you struggle with input or type conversions, you will have less time to devise and implement your solution!

Although this section will not be based around solving one large problem, it will teach you important mechanical skills you will need as you prepare for morning problems.

#### Learning Outcomes:

- Learn how your programs can take different types of keyboard input from a user while the program is running: strings, integers, floats, and lines containing space-separated input.
- Formally understand types in Python 3 and the purpose of type conversions.
- Become prepared for the input/output requirements of morning problems.
- Learn about some other built-in Python functions, including those specific to strings and lists, that will be useful to know in the future.

## 6.1 INPUT: STRINGS

### 6.1.1 THE INPUT FUNCTION

Python has a built-in function called `input`, used to prompt a user to type some keyboard input when the program is running. [You should read the documentation of the input function here.](#)

Open a new terminal and start the Python 3 interpreter. You can invoke the `input` function using:

```
>>> input()
```

You will see that the interpreter waits for you to type a line of input, which must end with the press of the **ENTER** key.

---

Here is a simple example. Try typing the following lines of code into the interpreter:

```
>>> a = input()
Hello
>>> print(a + " Goodbye")
```

You should see the string “Hello Goodbye” printed to the terminal window.

### 6.1.2 PROMPTING FOR INPUT

Sometimes, you may want to print a message to the screen to give the user some additional information about the type of input you are expecting. For example, say you want the user to enter their name. Here is one solution to this problem:

```
print("What is your name?")
name = input()
```

However, this results in the question being printed on a different line from the input prompt! Although we could solve parts of this problem with a little more work, this method still requires an extra line of code. A better way is to use the `input` function itself, which allows you to **include a prompt as an argument**:

```
>>> name = input("What is your name?")
What is your name?John
>>> print("Thank you", name)
Thank you John
```

Try typing this line for yourself to see how it works.

**Tip:** You may wish to add a `:` symbol and/or an extra space at the end of your prompt to make it more readable.

### 6.1.3 BUILT-IN STRING METHODS

Python has many built-in functions that can be used to perform operations on strings. Although this section will not explain all of these functions in detail, they are all useful to know and you are encouraged to take the time to look them up. While you do not need to memorize them all, you should know how to look them up when you need them to solve a problem!

Here is a brief table summarizing some of the most important string methods, with a direct link to the documentation of each. Several of them will be explained in the next sections, but the rest you can take the time to learn on your own.

<a href="#">str.capitalize</a>	<a href="#">str.count</a>	<a href="#">str.endswith</a>	<a href="#">str.format</a>	<a href="#">str.find</a>	<a href="#">str.isalpha</a>
<a href="#">str.islower</a>	<a href="#">str.isnumeric</a>	<a href="#">str.isspace</a>	<a href="#">str.isdigit</a>	<a href="#">str.isupper</a>	<a href="#">str.isalnum</a>
<a href="#">str.title</a>	<a href="#">str.lower</a>	<a href="#">str.strip</a>	<a href="#">str.replace</a>	<a href="#">str.startswith</a>	<a href="#">str.upper</a>

The “str.” prefix indicates that the function should be invoked as a **method** of a **string object**. You will learn more about this later when you are introduced to classes, but for now, know that if

---

you want to invoke a string method (for example the `islower` method on the string “Hello!”) you would do so like this:

```
"Hello".islower()
```

## Checking for Uppercase and Lowercase Characters

Some characters can be either uppercase or lowercase, such as the letters in the English alphabet. There are a number of string methods that check for or convert the [case of letters in a string](#).

For example, `isupper` checks that there is at least one uppercase character (and no lowercase characters) in a string:

```
# this will return False, because there is at least one lowercase character
"Hello".isupper()
# this will also return False
"10".isupper()
# this will return True, even with the 2!
"2B".isupper()
```

This method is nuanced; it only returns true if there are no lowercase letters in the string, and at least one uppercase letter! The `islower` method does the opposite, checking for lowercase letters instead.

Other methods involving letter case include `lower`, which converts all cased characters in a string to lowercase, `upper`, which converts to uppercase, and `title`, which converts the first letter of each word in a string to uppercase and all other letters to lowercase. Try the following in the Python3 interpreter to see what they do:

```
>>> "CMPUT274".lower()
>>> "hello world".upper()
>>> "this is a book title".title()
```

**Tip:** Before you move on, try all of the methods mentioned above in the interpreter. Read the documentation to learn how to invoke them correctly, and experiment. Become comfortable with at least some of the functions before you move on to the tasks of this section!

## Learn More

Here are several resources you can use to learn more about string methods:

- [This website](#) has an excellent list of all the string methods available in Python 3, with a description and examples of each.
- String Method Tutprials:
  - [A good general string tutorial and reference](#), especially for string formatting and escape characters.
  - [Another tutorial on string methods](#).

---

### 6.1.4 YOUR TASKS: STRING INPUT

Here are a few exercises to help you practice taking input and some basic string functions. Although you won't be solving any meaningful problems, these exercises are a good basic test of your knowledge of string functions and will get you comfortable taking strings as input. Try to complete as many as you can!

You can solve these problems in the interpreter, but you are encouraged to write .py files instead, as this will more closely mimic what you will do to solve a morning problem. For each problem, make sure to test an empty string "" as an input! This is an important **edge case** that will test if your program is working correctly. You will learn more about [edge cases](#) later.

#### Tasks:

1. Write a program that takes a string as input and prints the number of times the letter 'l' appears in the string to the screen. For example, "Hello" would print 2.
2. Write a program to determine if an input string contains the [substring](#) "abc". For example, the input "cabc" would output True because it contains the substring "abc", while the input "abd" would output False.
  - (a) Modify your program to instead determine if a string contains the number 6.
3. Write a program that takes a string and replaces all instances of the letter "c" with the letter "g". For example, "cake" would become "gake".
4. Write a program that checks if a string starts with an uppercase letter and ends with a lowercase letter, and prints either "Yes" or "No" accordingly.
5. Write a program that takes a string and prints only the first and last character of the string to the screen. For example, the input "Apple" would output "Ae" and "B" would output "B"  
**Hint:** You may need to check the string length to avoid indexing out of range!
  - (a) **Hard.** Write a second, similar program to print all letters in the string **except** the first and last. You can solve this problem without using a loop. **Hint:** [This page](#) may be helpful.

## 6.2 INPUT: INTEGERS OR FLOATS

You can use the `input` function to take in any line of input, including a number. For example:

```
6
2.4
-5
```

However, `input` **converts** the entire line it reads to the type **string**, (after stripping the trailing newline added by hitting the enter key). The issue is that if, for example, you take in an integer (such as the number 7) and attempt to perform mathematical operations on it, you will get a

---

TypeError! This is because the number 7 is actually considered the *string* “7” until it is **converted to an integer**.

For example, this will return a TypeError: “Can’t convert ‘int’ object to str implicitly”.

```
>>> a = input()
6
>>> a = a + 2
```

You see this error message because the Python interpreter first encounters the variable *a*, which points to an object of type *str*. The “+” operator is defined for strings as *string concatenation*, so the Python interpreter attempts to perform this operation. However, when it reaches the integer 2, it fails. The interpreter cannot do string concatenation on an object that is not of type *str*.

The interpreter cannot automatically (or **implicitly**) convert the type, because it does not know which operand to convert. Do you want to convert the integer 2 to string and do string concatenation? Or are you trying to add two integers? This decision is **ambiguous**.

To avoid this problem, in order to read in an object whose type is not string correctly, you must perform an **explicit type conversion** to tell the interpreter which operation you wish to perform.

### 6.2.1 INTRODUCTION TO TYPES IN PYTHON

All objects in Python 3 have a **data type**, which instructs the interpreter how the programmer intends to use the data. Sometimes, two objects can have the same representation in memory, even though they are intended for different uses. For example, there is a standard [ASCII table](#) of character codes used to represent symbols in memory. You can find an online copy of the table [here](#). Looking at it, you will see that the decimal number 65 (the far left column) maps to the character ‘A’. In memory, the binary representation of the integer 65 is used to represent both values. However, **the interpreter can use the type of the object in Python**, either *str* or *int*, to tell the difference.

You can use the builtin **type** function to check the type of an object. Its documentation is [here](#). For example:

```
>>> type(6)
<class 'int'>
```

The object belongs to a class called “int”.

In the interpreter, check the type of the following objects:

1. 2.5
2. "Hello World!"
3. print
4. [1,2,3]

---

### 6.2.2 TYPE CONVERSIONS

There are several simple functions that can be used to convert between basic types: `int`, `str`, `float` and `list`.

#### Integer Conversion

```
>>> int("6")
6
```

#### String Conversion

```
>>> str(123)
'123'
```

#### Float Conversion

```
>>> float(2)
2.0
```

#### List Conversion

Notice the interesting behaviour of converting a string into a list:

```
>>> list("Hello")
['H', 'e', 'l', 'l', 'o']
```

If you attempt to convert a value (or **literal**) that cannot be correctly converted, you will see a `ValueError`:

```
# this will cause an error - try it for yourself!
int("ABC")
```

### 6.2.3 WORKING WITH NUMBERS

Now that you have learned how to do basic type conversions, it is time to practice. In the exercises in the next section, you will learn how to use a few interesting and useful builtin mathematical functions: `abs`, `min`, and `max`. You can find the documentation of all three functions [on this page](#). You will need to learn on your own what each of the three functions do.

### 6.2.4 YOUR TASKS: NUMBER INPUT

Use the following exercises to help you discover the uses of `abs`, `min`, and `max`.

1. Write a program that asks the user to enter an integer. Then add 5 to the integer and print it to the screen.

2. Write a program that takes a float and if it is negative, converts it to a positive float before printing it to the screen. Do this **without** using a conditional statement.
3. Write a program that reads in two separate integers. Do this using two calls to `input` on different lines for now, unless you have read the next section. Output the higher of the two numbers.

(a) Write a second, similar program that prints the lower of the two numbers.

4. In Python, if you apply an integer type conversion to a floating point number, any digits right of the decimal point get removed. For example, `int(-2.5)` becomes `-2`, and `int(3.5)` becomes `3`. Therefore, the integer version of the number is the same as the **floor** of the float if it was positive, or the **ceiling** if it was negative. Conversion to integer can unintentionally reduce the precision of a float if you are not careful!

**Hard.** Write a program that takes a float as input. If the float is a whole number (eg, `-2.0` or `12.0`), convert it to an integer and print it to the screen. Otherwise, print a message telling the user that the conversion is not possible. **Your program should never truncate a float and should only perform the conversion if the number entered was a whole number.**

**Hint:** You can do this in multiple ways. One of them involves the `%` operator. Another uses string methods we have already learned.

## 6.3 INPUT: LISTS

A morning problem will often ask you to read a line of space separated integers like this:

```
>>> my_list = input()
1 2 3 4 5
>>> my_list
'1 2 3 4 5'
```

However, recall that the `input` function converts the entire line to a single string! You will need to use several different builtin functions in order to access the integers in this string directly.

The first thing you must do is **split** the integers by whitespace using the builtin function `split`.

```
>>> my_list = input().split()
1 2 3 4 5
>>> my_list
['1', '2', '3', '4', '5']
```

The `split` function (whose documentation can be found [here](#)) returns a list of the words in the string, where each word is separated by any amount of whitespace. If we wanted the integers to be of type `str`, then this would be enough! However, we want all the numbers to be of type integer so we can perform mathematical operations on them.

Fortunately, there is a builtin function for this as well: the `map` function. The [map function](#) takes an iterable (a list is an iterable) and applies a function to each element of the list. For example, we can convert all the numbers to type `int` using the `int` function like this:



```
>>> my_list = map(int, input().split())
1 2 3 4 5
>>> my_list
<map object at 0x7f010e45acc0>
```

Well, that didn't quite work as expected. The `map` function creates a map object. In order to easily access the integers, we need to convert the map object back into a list:

```
>>> my_list = list(map(int, input().split()))
1 2 3 4 5
>>> my_list
[1, 2, 3, 4, 5]
```

Now our list is correct and we can use its elements as integers. This is among the most complicated types of input, but you will be expected to read in lines of integers in almost every morning problem.

**Sidenote:** There is another method that does the same thing as the above, using a fancy trick called [list comprehension](#):

```
>>> my_list = [int(i) for i in input().split()]
1 2 3 4 5
>>> my_list
[1, 2, 3, 4, 5]
```

As above, this method converts each element in the list `input().split()` to an integer. This is an abbreviated version of a **for loop**, which you will learn in the next lab.

### 6.3.1 WORKING WITH LISTS

This section will teach you some important operations you can use when working with lists, including adding or removing elements of a list, reversing a list, and getting the smallest or largest element of a list.

[This page](#) gives a short, excellent summary of all of the basic operations that can be done on lists. It is well worth using as a reference to understand list functions.

#### Append, Extend, and Insert

To add an element or elements to a list, there are three different functions: `append`, `extend`, or `insert`.

If you want to add a single element to the end of a list, use `append`:

```
>>> my_list = [1,2,3]
>>> my_list.append(1)
>>> my_list
[1, 2, 3, 1]
```

---

Notice that you do not use an = sign for this! The **append** method modifies the list **my\_list** **in-place**. This means that the object referred to by the identifier **my\_list** was changed by the method, even without an assignment operator.

If you wish to concatenate two lists, you can use **append** to add the whole list as an element:

```
>>> my_list = [1,2,3]
>>> my_list.append([4,5,6])
>>> my_list
[1, 2, 3, [4, 5, 6]]
```

or use **extend** to append each element of the second list to the first one:

```
>>> my_list = [1,2,3]
>>> my_list.extend([4,5,6])
>>> my_list
[1, 2, 3, 4, 5, 6]
```

Note that using the + operator (which can also be used to concatenate two lists) produces the same result as **extend**.

Finally, you may want to add a new element into a list at an arbitrary position (since **append** always places a new element at the end of the list). For this, you can use **insert**:

```
>>> my_list = [1,1,1]
>>> my_list.insert(2, "a")
>>> my_list
[1, 1, 'a', 1]
```

The **insert** method takes two parameters. The first is an **index** where a new element should be inserted, and the second is the element you wish to insert.

## Deleting Elements

You can delete an element using the **del** keyword. To delete an element of a list:

```
my_list = ['a', 'b', 'c']
```

type the keyword **del** followed by a space, and then the element you wish to delete. You must specify the element like this:

```
# this will remove the first element, at index 0
del my_list[0]
```

Other functions you can use to remove an element of a list are **pop** and **remove**, which you can research on your own if you are interested.

## Smallest and Largest Elements

You can use the builtin functions **max** and **min** to get the largest or smallest element of a list:

```
>>> my_list = [2, 4, 5, 1, 6, 8]
>>> max(my_list)
8
>>> min(my_list)
1
```

## Reversing a List

The `reverse` method can be used to reverse the order of elements in a list. For example:

```
>>> my_list = [2, 4, 5, 1, 6, 8]
>>> my_list.reverse()
>>> my_list
[8, 6, 1, 5, 4, 2]
```

Notice that the `reverse` method also modifies the list in-place.

### 6.3.2 YOUR TASKS: LIST INPUT

1. Create an expression that will read in a line of space-separated floats. How does the expression change from the one used to read a line of integers?
2. Create a program that will take a space-separated list of integers as input and perform the following operations in order:
  - (a) Reverse the list.
  - (b) Append the number 3 to the list.
  - (c) Delete the first element of the list.
  - (d) Append each element in the [1,2,3] to the list, in order.
  - (e) Append a new element to the list that is equal to the sum of the first two elements.
  - (f) Reverse the list again.
  - (g) Print the final list to the terminal screen.

For example, the input “1 2 3 4 5” should produce the output:

```
[7, 3, 2, 1, 3, 1, 2, 3, 4]
```

3. Write a program that reads a line containing  $n$  integers, where  $n$  is an arbitrary positive integer, and then outputs the largest number. For example, this input should print 5 to the terminal.

```
python3 largest_num.py
-5 -2 4 -12 -9 5 -7 1
```

4. Write a program that takes a list of space-separated integers, floats, or strings and switches the first and last element. You are guaranteed that the input list will have at least two elements. For example, the list [1,2,4,3] would become [3,2,4,1]. **Hint:** There are at least two different ways to solve this problem. One involves something called **multiple assignment** in Python which you will have to look up, while the other does not. Try to implement both methods.

---

## 6.4 OUTPUT TRICKS

Every morning problem will expect you to read in some input and output a result. Usually, the result will be a single value, and the `print` function alone will suffice.

Other times, it is necessary to work a little harder to produce the correct output. Your solution will be considered incorrect unless the output matches the expected result exactly, so this is important! The next sections will teach you a few common scenarios that you may encounter when formatting output in a morning problem.

### Outputting a Space Separated List of Elements

Here is a sample output you may need to reproduce in a morning problem situation:

```
# the answer is:
[7, 3, 2, 1, 3, 1, 2, 3, 4]
# but it must be formatted as:
7 3 2 1 3 1 2 3 4
```

In this situation, you can use a built-in method called `str.join` ([documentation here](#)). You can use `join` to concatenate all the elements of a list. The string `str` is the character or characters that will act as the separator between each list element.

**Important Note:** For this to work, all the elements in the list must be of type `string`. Otherwise, you will get an error when you attempt to join the integers together.

For example:

```
my_list = [1, 2, 3, 4, 5]
# note that each element must be a string in order for this to work
# so we must map all elements in the list into a string
my_list = list(map(str, my_list))

# this creates the string "12345"
my_string = "".join(my_list)

# this creates the string "1a2a3a4a5a"
my_string = "a".join(my_list)

# and finally, a space-separated string:
my_string = " ".join(my_list)
```

Another method uses a `for` loop (which you will learn in lab 5) in order to construct a string that can then be printed:

```
my_list = [1, 2, 3, 4, 5]
print_string = ""

# this loop iterates over each element of the list
for element in my_list:
```

```

# we must still convert the element to a string
print_string += str(element)
# add a space after each element, including the last one!
print_string += " "

# now print_string holds the string
# "1 2 3 4 5 "
# (note the extra space)

# print the string
# the strip function removes the extra whitespace at the end first
print(print_string.strip())

```

**Sidenote:** There is another method that does the same thing as the above, using the `*` operator. You have seen `*` for multiplication and exponentiation, but it can also be used to [unpack the elements of a list](#):

```

>>> my_list = [1, 2, 3, 4, 5]
>>> print(*my_list)
1 2 3 4 5

```

This operator **unpacks** the values in the list and treats each as a separate object. Then, the print function interprets them as if they were individual, comma-separated elements. **Note:** Although this method is perhaps the fastest way to print the space-separated elements of a list, it is not the simplest or the most understandable.

## Note: Augmented Assignment Operators

In the code above I used an operator you have not seen before: `+=`. This is an example of an **augmented assignment operator**, which combines a binary operator such as `+`, `-`, `*`, or `\` with an assignment operator `=`.

```

x = 5
x += 1      # equivalent to x = x + 1
x *= 3      # equivalent to x = x * 3
x /= 2      # equivalent to x = x / 2
x -= 4      # equivalent to x = x - 4

```

Augmented assignment operators are essentially a shorthand notation. Although both methods obtain the same result, there are subtle differences in the way they are evaluated. If you are interested to learn more, you can read these pages: [Augmented Assignment statements](#) and [Python FAQ: Why does adding an item to a tuple raise an exception when the addition works?](#) (The second link is quite advanced).

## String Formatting

Here is a quick introduction to string formatting. Say you need to print a variable in the middle of a string, like this:

```

‘‘Hello, NAME! How are you?’’

```

---

where you wish to replace NAME with a variable called `your_name`.

To do this, there are two different methods. One of them uses a method of string formatting similar to that of the C programming language, which we will not cover here. This section will show you a different method using the `str.format` function.

Using this method, you can place an empty pair of curly braces wherever you would like a variable to be inserted, and then place a comma-separated list of variables as the arguments to `str.format`

```
s = 'Hello, {}! How are you?'.format(your_name)
```

You can place as many variables into the string as you like:

```
s = 'Milk: {}, Eggs: {}, Bread: {}'.format(milk_qty, egg_qty, bread_qty)
```

There are other, more advanced operations you can do to format strings using `str.format`. See the additional output resources section below if you are curious to learn more.

## Additional Output Resources

Here is a list of webpages that you can reference when formatting output:

- [Formatted Output: A good general string formatting reference.](#)
- [A nice tutorial on using the str.format function, including advanced applications.](#)

## 6.5 YOUR TASK: DATE CONVERTER

Create a program that reads in a space separated date in the format:

```
year month day
```

and outputs the date in the format:

```
month.day.year
```

For example, the input “1998 Febuary 3rd” would output as “Febuary.3rd.1998”

For an additional challenge, format the date into an output string as: “The new date format is Febuary.3rd.1998.”. Don’t forget the dot at the end!

## SECTION 7

---

### LAB 5 - ITERATION: CLOCKS, FAST AND SLOW

---

The problems you have solved so far have all been relatively simple. Before now, you had not been introduced to a key programming concept: **iteration**. Iteration, or looping, allows a program or a section of a program to *repeat multiple times*, or even *loop indefinitely*. You might use iteration to *continue taking input* from the user until the user types the phrase “quit”, or to *loop over* all the elements in a list or all the characters in a string.

Iteration is key to almost every meaningful problem you will solve as a computing scientist or computer engineer.

#### Learning Outcomes:

- Understand definite and indefinite iteration, and when each might be useful.
- Learn basic modular arithmetic to solve a simple problem involving clocks.
- Introduction to a problem-solving strategy: simulation.

### 7.1 CLOCKS, FAST AND SLOW

In this lab, you will learn how iteration works in order to solve a problem involving two clocks. Here is the description:

“You have two 12-hour clocks. One is two minutes slow, losing two minutes every hour and the other is two minutes fast, gaining two minutes every hour. If both clocks start showing 12:00, how long will it take until both clocks again display the same time?”

In order to solve this problem, you will need to know several new concepts, including basic iteration and modular arithmetic.

### 7.2 LOOPS

Imagine that you want to print the string “Hello World!” five times. You might do this:

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

---

Now imagine that you wanted to print “Hello World!” one hundred times. Or one thousand. Typing (or even pasting) that many print statements is just not feasible.

In another scenario, imagine you want to print all the integers, starting from 1, one by one, until you reach a user’s favourite number. You could do this:

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

and continue typing print statements until you reached the number specified. However, you would need to know the user’s favourite number in advance order to include enough print statements, and it would be easy to make a mistake.

A **loop** is useful when you want to repeat a task multiple times. Code within a loop structure will continue executing over and over again until a certain condition is met. Each full execution of the code within a loop is called an **iteration**. The number of iterations might be known in advance (ie, print “Hello World!” five times) or unknown until the loop is executing (print all numbers up to a favourite number). In another case, such as a simple guessing game, you might keep asking the user for input until they guess your favourite number.

Loops help to shorten and **modularize** code. With a loop, you need only type the body of code once, which prevents many lines of code. Additionally, the code is modular; the loop code is contained within the loop structure, forming its own section.

In Python, there are two types of loops: **for loops** and **while loops**.

### 7.2.1 FOR LOOPS

A for loop is useful when

1. You know exactly how many iterations you will need to complete in advance.
2. You want to **iterate over** all the elements of a list, string, or other container and complete the same operation for each one.

Here is the syntax of a for loop in Python:

```
for (iterating variable) in (sequence):
    (do something)
```

Often, the iterating variable is often replaced by the letter **i**, short for **iterator**. This is simply a convention; you can call the variable whatever you like. All of the code within a loop in Python must be **indented** in order to be considered part of the loop.



---

## The Range Function

If you need to iterate over a sequence of numbers, the `range` function ([documentation here](#)) is extremely useful. For example:

```
for i in range(0, 5):  
    print(i)
```

This loop will produce the output:

```
0  
1  
2  
3  
4
```

Notice that we type `print(i)` rather than printing any specific number. In each iteration, `i` is set to the numbers 0 through 4 in the range.

The range function creates a **range object** containing all the integers  $x$  where  $0 \leq x < 5$ . Note that the **lower bound** (the first parameter, 0) is included in the range while the **upper bound** (the second parameter, 5) is not included.

To give another example, if you wanted to print the numbers from 1 to 10 using range, you would need to do this:

```
for i in range(1, 11):  
    print(i)
```

because 1 is included in the range while 11 is not.

There are many more advanced applications of the range function. For example, the range function can accept an **optional third parameter**, which is the **step**. The step is set to 1 by default, so if no step is specified, the range function will simply use the value of 1.

The step determines how many numbers to “jump over” when creating the range. For example:

```
# 1. This range has a step of 1 and produces the range:  
# 1, 2, 3, 4, 5  
range(1, 6, 1)  
# 2. This range has a step of 2 and produces the range:  
# 1, 3, 5  
range(1, 6, 2)
```

You can also iterate **backwards** by using a **negative step**.

```
for i in range(10, 0, -1):  
    print(i)
```

Notice that the old rules still apply! The first parameter (10, which is now an upper bound) is included while the second (0, the lower bound) is not. Try this loop for yourself in the interpreter to see what it does.

---

## Iterating Over List or String Elements

In this section, you will learn two different ways to iterate over a container in Python.

If we want to iterate through a range of values in Python, say for example the numbers from 1 to 10, the simplest way is to use Python's **in keyword**. The most basic use of **in** is to check if an element is in a container. For example:

```
>>> my_list = [2, 6, 12, 56, 23, 1, -5, 2]
>>> 5 in my_list
False
>>> 2 in my_list
True
```

However, the **in** keyword can also be used to traverse through a sequence in Python. In the previous section, we used **in** to iterate over each integer in a range. Now, we will use it to iterate over each element in a list.

### Python 3: Reserved Keywords

Keywords are words reserved by the Python language for a special purpose. For example, **True** and **False** are keywords reserved for storing the logical values of true and false. Some other keywords you have already seen are **if**, **or**, **and**, **del**, and **for**. If you are curious about keywords in Python and their various purposes, [this tutorial](#) is a great place to learn more.

For example, assume we have the following list of integers and we wish to add 5 to each element. In loop terms, we want to iterate over all the elements, and **for each element in the list**, we want to add 5 to that element. To do this, there is an intuitive for loop structure:

```
my_list = [2, 6, 12, 56, 23, 1, -5, 2]

for i in my_list:
    print(i + 5)
```

This for loop iterates through each element **in** the list, adds 5 to that element, and prints the result. Note that this will not update any of the list elements, since **i** is a temporary variable that is set to each element in the list before each iteration of the loop.

We can also iterate over the characters in a string:

```
my_string = "Hello World!"
upper_string = ""

# iterate over each character in the string
for char in my_string:
    if char.isupper():
        # if the character is uppercase, add it to a new string
        upper_string += char

# print the new string of uppercase characters
print(upper_string)
```

---

This simple program finds all the uppercase letters in the string "Hello World" and concatenates them into a new string called `upper_string`. This new string will contain the letters "HW". Notice that in this program, the iterating variable is called "char", short for character, instead of `i`. This was done to better describe what the variable represents during each iteration.

## Another Way to Iterate Through a Sequence

Although the syntax above is very intuitive, it is unique to high-level languages like Python. In a language like C or C++, which you will learn later, the `in` keyword does not exist. In order to iterate through a sequence, you must remember a key fact: each element in the sequence has an index. By iterating over the indices, you can access each element in the sequence in turn.

In Python, using this alternate method, a loop that adds 5 to each element of a list would look like this. Notice that we are iterating through an index from 0 to the final index of the list (its length minus one):

```
my_list = [2, 6, 12, 56, 23, 1, -5, 2]

for i in range(0, len(my_list)):
    my_list[i] = my_list[i] + 5
```

Although creating a loop this way may seem confusing and unnecessary, it is mentioned here for three reasons. First, it is important to remember that each element of a list or string also has an index. Second, in some programming languages, manipulating these indices is the only way to iterate through a container like this, so it is an important method to keep in mind for later. Finally, it also provides a simple way to access the previous or next element in the list. If you access `my_list[i]`, you can also access `my_list[i-1]` and `my_list[i+1]` easily.

### 7.2.2 WHILE LOOPS

A **while loop** is the second type of loop in Python. A while loop will continue iterating indefinitely **as long as an initial condition is true**. Here is the basic syntax of a while loop:

```
while (a condition is True):
    (do something)
```

This condition can be any normal condition you have seen in the past. For example:

```
x = 0
# the loop will execute as long as x is less than 5
while x < 5:
    print(x)
    # the following is augmented assignment: equivalent to x = x + 1
    x += 1
```

The while loop above executes in exactly the same way as this for loop:

```
for x in range(0, 5):
    print(x)
```

---

A for loop is best when the number of iterations is known in advance. In the example above, both loops execute exactly 5 times each, and the while loop simply simulates the for loop.

## Indefinite Iteration and Break Statements

A while loop is useful when

1. You do not know ahead of time when your loop will end.
2. You are waiting for an event that could happen in an unknown number of iterations.

In other words, while loops are best for **indefinite iteration**, which is often implemented using an **infinite loop**. Typically, we create an infinite loop by setting the condition of the while loop to True:

```
while True:
    print("Hello World!")
```

The above loop will continue printing “Hello World!” until we force the process to terminate using CTRL+C. Infinite loops are useful when you want to execute code until a certain triggering event occurs. For example, this program asks a question and continues asking until the user enters the correct answer:

```
while True:
    name = input("Enter the name of this course: ")

    if name == "CMPUT 274":
        print("That is correct!")
        break

    else:
        print("Wrong answer. Try again!")
```

To exit an infinite loop (or to prematurely leave a loop) we use the **break statement**. This is a special keyword that stops execution of a loop and skips to the next statement outside of the loop. In the above program, we used a break statement to stop looping once the user entered the correct answer. See [this tutorial on break statements](#) for an excellent flow diagram that illustrates what a break statement does.

## 7.3 MODULAR ARITHMETIC AND THE MODULO OPERATOR

Readings on Modular Arithmetic:

- [Better Explained: Fun With Modular Arithmetic](#). This is a very good, thorough explanation in simple terms. It also includes some uses for modular arithmetic and is generally an interesting read.
- [Modular Arithmetic and the Multiplicative Cipher](#): This is a chapter of an introductory cryptography textbook that begins with a good introduction to modular arithmetic.

- 
- [Wikipedia: Modular arithmetic](#).

Time keeps going on and on (possibly forever), but as you’ve probably noticed, clocks don’t have an infinite number of numbers on them. On a normal 12-hour clock, the time goes from 1 to 2, then 2 to 3, then 3 to 4, and so on, up until it goes from 11 to 12, and then from 12 back to 1 again. In other words, we use a system with only 12 numbers for keeping track of hours of time. After the clock’s hand swings around those 12 hours, it starts over at 1 again.

This method of cyclical counting (wrapping time around a circular clock) can be described in terms of a type of math called modular arithmetic. Modular arithmetic is a system of math for integers that involves numbers that “wrap around” in a predetermined cycle. In modular arithmetic, the number line is not straight and infinitely long; instead, it wraps around in the shape of a circle. Later on in CMPUT 274, you will learn more about modular arithmetic using C++ and the Arduino. For now, you only need to know basic clock arithmetic and the modulo operator in Python 3.

The length of the circular number line is called the **modulus**. For a 12 hour clock, the modulus is 12, because there are 12 numbers on the clock. We know that when it is 9 o’clock, 8 hours later it will be 5 o’clock, even though  $9 + 8 = 17$ .

### 7.3.1 THE MODULO OPERATOR

In modular arithmetic with positive numbers, we deal in **remainders** between 0 and the modulus (not including the modulus). Programming languages (including Python) typically support a modulo (often shortened to “mod”) operation that will return the remainder of a division.

In Python, we use the symbol `%` for this operation. Open the interpreter and try each of these examples to see how this operation works:

```
>>> 2 % 2
>>> 4 % 2
>>> 121 % 2
>>> 13 % 3
>>> 17 % 12
>>> 5 % 12
```

In each case, the result of `n % m` is the same as if we had wrapped the first number, `n`, around a clock with `m` numbers. For example, if we wrap 13 around a clock with 3 numbers (0, 1, and 2), we wrap 4 times and arrive at the number 1. This is also the remainder of  $13/3$ .

## 7.4 YOUR TASKS: LOOPS AND THE MODULO OPERATOR

This section is split into two parts. If you are struggling with loops, complete the following simple practice exercises first before moving on to the main exercises of this section:

### 7.4.1 SIMPLE TASKS:

1. Create a new loop for each of the following tasks. For the numbers between 1 and 10 (inclusive):

- 
- Print only the even numbers.
  - Print only the odd numbers.
  - Print the numbers in reverse from 10 to 1.
  - **Hard.** Print all the numbers on the same line, instead of on separate lines.
2. Create a list with 5 integers (you may alternately allow the user of your program to enter the list as input). Create a new loop for each of the following tasks:
    - Print only the odd integers in the list.
    - Convert each integer to a string, multiply it by 2, and convert it back to an integer. For example, the integer 1 would become the string “1” and then when doubled would become “11” and then the integer 11.
  3. Given a string as input:
    - Use a loop to check if the string contains the letter “i” in either upper or lower case. When you find the letter, your program should immediately stop looping, print a confirmation message, and exit. Your program cannot use the `in` keyword or the `str.find` method or any other builtin string methods to find the “i”.
    - Use a loop to count the number of characters in the string. Do not use the `len` function. **Hint:** Try setting a character count to 0 before the loop, and then modify it somehow in each iteration.

#### 7.4.2 MAIN TASKS:

1. Make a number guessing game. Generate a random number from 1 to 100 (or any range you like). Prompt the user to enter a guess between 1 and 100. If their guess is wrong, tell them whether the guess is too high or too low, and ask them to guess again. The user should be allowed to keep guessing until they have successfully guessed the number.
2. Write a program that prints all numbers from 1 to 100 (inclusive) that are multiples of 3 but *not* multiples of 5. For example, the first few outputs of your program should be: 3, 6, 9, 12, 18. Note that 15 is not included because it is also a multiple of 5.
3. The above problem is a variation on the [FizzBuzz test found here](#), which is also worth implementing if you are a new programmer. It is considered a classic programming interview question. Here is the description:

“Write a program that prints the numbers from 1 to 100. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz”.”

Implement a solution to this problem. The link above may give you some ideas if you are stuck.
4. **Hard.** Read a list of space-separated integers from the user. Multiply each element in the list by the previous element. So for example, the second element in `your_list` would be: `your_list[1] = your_list[1] * your_list[0]`. The first element should not be multiplied (as there is no previous element), but should still be included in the output. Note: you will

---

need to find a way to access each previous element! When you are done, print the list in the same format as the input list (space-separated) to the screen. For example, with the input:

```
1 2 3 4 5
```

Your program should output:

```
1 2 6 12 20
```

5. **Hard.** The string function `str.title` converts a string of words such that each word starts with an uppercase letter and makes all other letters lowercase. Implement this function yourself using a loop. You can use the functions `str.upper` and `str.lower` to help you. **Hint:** Consider the following pseudocode to help you solve the problem:

```
for each letter in the string:
    if (this letter is the first letter in a word):
        set it to uppercase
    else:
        set it to lowercase
```

It is up to you to devise a strategy to determine what makes a letter the first one in a word.

## 7.5 NESTED LOOPS

Sometimes, you may need to execute multiple loops inside of each other. For example, one of the challenge exercises at the end of this section may require you to use nested loops. The basic structure of a nested loop looks like this:

```
loop 1:
    body of loop 1

    loop 2:
        body of loop 2

    body of loop 1

code outside of loop
```

Notice that the second loop must be indented within the first loop in order for it to be properly recognized and executed by the Python interpreter.

**Here are some examples:**

1. You have two lists of numbers and you want to try all possible combinations (taking one from each list).

For example, given two lists, you want to try all combinations to see how many of them sum to the number 7:

```
list1 = [1,2,3,4,5]
list2 = [2,3,4,4,5]
```

```

count = 0
for i in list1:
    print("Running outer loop")
    for j in list2:
        print("Inner loop")
        if i + j == 7:
            count += 1

print(count)

```

A loop structure like this is commonly called a **double for loop**. Try this code for yourself to see how it works. Notice how the outer loop runs only once for every five iterations in the inner loop (given these two example lists). Try modifying the elements in the loop to see how the final result changes.

2. Here is a second example, using two nested while loops to find all prime numbers from 2 to 100:

```

i = 2
while(i < 100):
    j = 2
    prime = True
    while(j < i):
        # if i is divisible by any number from 2 to i-1, it is not a prime
        # (we could be more efficient by checking only up to i//j
        # but this is more complicated)
        if (i%j == 0):
            prime = False
            j = j + 1
    if prime == True:
        print(i, "is prime")
    i = i + 1

print("Good bye!")

```

We can use two for loops to do the same thing:

```

for i in range(2, 101):
    # for each number, we assume it is prime by default
    prime = True
    for j in range(2, i):
        # if i is divisible by any number from 2 to i-1, it is not a prime
        if (i%j == 0):
            prime = False
    if prime == True:
        print(i, "is prime")

print("Good bye!")

```

Checking the **primality** of a number is an important concept to fields of computing science such as cryptography and data security. We will discuss prime numbers and some related algorithms in more detail in future labs.



---

## More Resources on Nested Loops

- [Python 3: Nested Loops](#). This tutorial contains another example: generating multiplication tables using nested loops.
- [How to Construct For Loops in Python 3](#). Near the bottom of this page, there is a section on nested loops that gives examples showing how to iterate through lists composed of lists. The rest of the page is also a good review of `for` loops in general.

## 7.6 YOUR TASK: SOLVING THE FAST AND SLOW CLOCKS PROBLEM

Write a program to solve the Fast and Slow Clocks problem presented at the beginning of this lab. Here is the description again:

“You have two 12-hour clocks. One is two minutes slow, losing two minutes every hour and the other is two minutes fast, gaining two minutes every hour. If both clocks start showing 12:00, how long will it take until both clocks again display the same time?”

### 7.6.1 SIMULATING CLOCKS

In order to solve this problem, we will use a strategy of **simulation**. Our Python program will **simulate** running the clocks as long as it takes for them to again display the same time.

This is a simple strategy. We will use an infinite loop to continually update the real time and the time passed for the fast and slow clocks. Eventually, when the clocks at some point show the same time, we can break out of the loop to display the final answer and exit.

Sometimes, it is possible to solve the problem in a much shorter way by creating a [closed-form expression](#). If we could find an algebraic expression that arrived at the correct answer without using any loops, for any inputs, it would execute much faster than our simulation algorithm. However, it is often more difficult to come up with closed-form expressions. In this case, we will use simulation because it is simpler to understand and implement, and because our simulation does not take noticeable time to complete.

### 7.6.2 SOLVING THE PROBLEM

Although you will not be given any starting code for this problem, you can follow this series of tips if you need help:

1. The slow clock will increase by 62 minutes for every hour that passes in real time, and the fast clock will increase by 58 minutes.
2. Use an infinite loop (indefinite iteration) to loop until the times on the two clocks are the same. The basic structure of your program (in pseudocode and with comments describing the sections) might look like this:

```

# declare your variables here
# including the start times of the slow and fast clock
# the time elapsed in real hours should start at 0

while True:
    # update the slow clock
    # update the fast clock
    # update the time elapsed in real hours

    # your equality check will likely be more complicated than this!
    # you have to keep track of time in both hours and minutes
    if (slow clock time) == (fast clock time):
        break

# output the number of hours it took to get here (the real time)

```

3. You can think of each iteration of your main loop as being equivalent to one hour passing “in real time”.
4. If you are struggling, add a print statement in each iteration of the loop so that you can see what your program is doing at each iteration. Consider printing the updated time of both clocks during each loop.

### 7.6.3 CHALLENGE EXERCISES

If you have solved the basic exercise, try these harder ones:

1. Abstract your program in the following way. Instead of the clocks always being two minutes slow and two minutes fast, change your program so it takes two integers on a single line as input. The first integer is the number of minutes lost by the slow clock every hour and the second is the number of minutes gained by the fast clock every hour. Both numbers will always be positive. For example, the input to create the initial problem is:

```
2 2
```

2. **Hard.** Modify your program so that instead of taking the modifiers as a line of input, your program iterates through **every combination** of modifiers from (0,0) up to (60, 60). For each combination, determine how long it takes for the clocks to again show the same time. Try to answer the following questions:
  - (a) What is the longest amount of time two clocks will take to catch up?
  - (b) What is the shortest amount of time?
  - (c) Which fast and slow modifiers produce these outputs, and why do you think this is?

## SECTION 8

---

### LAB 6 - PROBLEM-SOLVING AND ALGORITHMS

---

By completing the first four Python-related labs, you have gained a basic set of programming tools: variables, conditionals, input and output, strings, lists, and loops. Now that you have the necessary Python knowledge, it is time to learn about algorithms and problem-solving.

#### Learning Outcomes

- Understand the purpose of an algorithm.
- Look ahead at some of the algorithmic paradigms you will learn in CMPUT 274/275.
- Learn the steps to devising an algorithm to solve a specific problem.
- Practice devising algorithms for and solving several problems from scratch, given only English descriptions of the problem.

#### 8.1 WHAT IS AN ALGORITHM?

According to [Wikipedia](#), an algorithm is “an unambiguous specification of how to solve a class of problems”. Let’s break that down.

An **algorithm** is a step-by-step process that describes how to solve a problem and/or complete a task. It must be **unambiguous** in order to clearly and precisely specify a method of finding a solution. It must apply to a **class of problems**, meaning that the method must be general enough to work for a range of similar problems. For example, a method to sort a list of integers must work for **all lists of any integers** in order to be considered an algorithm. If it only works for one list of integers, one time, it is not an algorithm.

An algorithm is like a **recipe** or an instruction set, a well-defined procedure that lets a computer solve a problem. However, an algorithm is not the same thing as a program. An algorithm is **independent of any particular program or programming language**. We write a program to **implement**, or carry out, the steps of an algorithm.

##### 8.1.1 SNEAK PEEK: ALGORITHMIC PARADIGMS OF CMPUT 274 AND 275

There are a number of classic algorithmic paradigms that you will learn over the course of CMPUT 274 and 275. By the end of the two courses, you should be familiar with most or all of the following algorithm types. For now, you only need to understand brute force and simulation. The rest are a sneak peek at what you will see later on.

---

**Brute Force.** This is the most basic type of problem-solving strategy. At its simplest, brute force can be boiled down to: “try everything until something works.”. It is often the most straight-forward approach to exploring a solution space, but can take an unreasonable amount of time for large data sets. Trying every possible combination of digits, letters, and symbols to find a correct password is an example of a brute force algorithm, but one that can take years to crack a long password. There is an entire field of computing science dedicated to finding ways to improve algorithms so that they become *faster* than brute force.

**Simulation.** Not exactly the same as brute force, a simulation algorithm iteratively solves a problem by completing the required steps in the same order as would solve the problem by hand. You saw an example of a simulation algorithm when you solved the Clocks: Fast and Slow problem from , when you simulated the passage of time for two clocks.

**Recursion.** A recursive algorithm typically involves a **function calling itself**. So far, you have seen functions such as `print`, `input`, and `len`, but in the future you will learn how to define your own functions. In a recursive algorithm, a function continues calling itself (often with slightly modified parameters each time) until a desired outcome is reached. The key idea is to solve small problems and use the solution found to solve a larger, original problem. Recursion is a strategy used in other algorithm classes, from dynamic programming to some sorting algorithms. Here is [a video on recursion](#) if you are curious.

**Sorting.** There are a wide variety of algorithms used to sort data sets (such as a list of numbers). Each sort is useful for a different purpose, depending on the size of the data set, how close the data is to already being sorted, and other factors. Additionally, sorting algorithms run at different speeds, and some are faster than others. Sorting algorithms that you will learn in this course include **bubble sort**, **merge sort**, **quicksort**, and **selection sort**.

**If you are curious:** Here are some extremely interesting simulations of various sorting algorithms, showing each value as a vertical or horizontal bar with a different length. Even if you do not yet understand all of the sorting algorithms, the videos are still interesting to watch. It is possible to sort data in many different ways!

- [This website](#) allows you to simulate any one (or all) of 8 different sorting algorithms. At the bottom of the page, you can modify the parameters and observe the algorithms on lists that are random, nearly sorted, reversed, or have only a few unique values.
- [This YouTube video](#) visualizes 16 different sorting algorithms on a colour wheel and (personal opinion) is very satisfying to watch.

**Graphs.** Sometimes, it is possible to model data in the form of [a graph](#) with **vertices** and **edges**. An example might be a city map, where the vertices are intersections and the edges are the roads connecting them. There are a number of algorithms used to traverse graphs. A common problem that you will tackle in this course involves finding the **shortest path** between two vertices. Some graph algorithms that you will learn later include **breadth-first search**, **depth-first search**, and **Dijkstra’s algorithm**.

**Dynamic Programming.** A DP algorithm is like faster brute force. You will learn how to save calculations using a list, matrix, or dictionary-like structure to minimize the amount of execution time. **Top-down DP algorithms** use recursion and dictionaries (or maps in C++), while **bottom-up DP algorithms** use lists (arrays in C++) or matrices to store the data.

---

**Divide and Conquer.** Some problems need to be solved using a divide and conquer approach. Sometimes, although the whole problem is too large, we can solve a **subproblem** easily. In a divide and conquer algorithm, the problem is **divided** into manageable parts and then **conquers** each part to achieve the desired solution.

**Binary Search.** This type of algorithm is a form of divide and conquer in which you **search** for a value by continually splitting a sequence and narrowing down your search until the value is found. Binary search requires the input data to be pre-sorted in ascending or descending order.

### 8.1.2 THE IMPORTANCE OF DATA STRUCTURES

An algorithm is useless without an appropriate **data structure**. It is possible to store data in many different ways. So far, you have seen **variables, lists, and strings**. However, Python has many other data structures you will learn later on, including **dictionaries, sets, tuples, and classes**. The way that data is stored can affect not only which algorithm you use to solve a problem, but also how effective or efficient that algorithm can be. For example, without a sorted sequence, a binary search is an ineffective search algorithm. This concept, that data structures and algorithms go hand-in-hand, is an important one to keep in mind as you continue in CMPUT 274 and 275.

### 8.1.3 LEARNING MORE ABOUT ALGORITHMS

As you complete CMPUT 274 and 275, you will become familiar with many more algorithms. However, if you want a head start or are curious to learn more, the following links are a good place to begin:

1. This [chapter of the Computing Science Field Guide](#) gives a thorough introduction to algorithms, including the distinction between an **algorithm** and a **program** and an introduction to **complexity**.
2. This [Khan Academy tutorial on algorithms](#) is a good place to learn a number of basic algorithms. The introductory video is linked to here.

## 8.2 DEVISING A STRATEGY TO SOLVE YOUR PROBLEM

So how do you go about choosing or creating an algorithm to solve your specific problem?

The first step is to **understand the problem**. If you are working from a written problem description, read it several times and make sure you understand each part. Sometimes you may need to do some research to understand the vocabulary if new terms are introduced. You have to know the problem before you can solve it.

When you understand the task, **break it down** into manageable sections. What are the main parts of the problem? Are there easier **subtasks** that can be solved individually?

Next, **solve the problem at least once by hand**. Walk through the problem exactly as if you were a computer. What are the exact steps you must take to reach the solution? **Write down the steps as you determine them**. These steps are your algorithm to solve the problem. This step

is often overlooked, but is extremely important because it allows you to thoroughly understand how an algorithm would solve the problem.

Now that you have a general plan, **determine how you will store the data**, what structures you will need, and what you will use to implement the steps of the algorithm (selection, iteration, etc).

Make sure you complete these steps before you touch any code. **Never code blind!** Know what you are doing before you get started, and go in with a plan. You need to know your algorithm before you can implement it.

### 8.2.1 EXAMPLE:

I want to create a program that will determine who has won a Rock Paper Scissors tournament ([rules found here if you need a refresher](#)). The input consists of a single line of space-separated matches. For example:

RP SR PS RS SR PP PR SR

where R stands for rock, P stands for paper, and S for scissors. In each match, the first letter was Player 1's move, while the second letter was Player 2's move. To solve the problem, I want to output either "Player 1", "Player 2", or "Tie", depending on the outcome of the tournament overall.

The above is an English description of the problem. Here is how I would go about solving it:

1. **Understand the problem.** I reread the problem description and take a look at the rules of rock paper scissors to make sure I understand them. I pay special attention to the fact that it doesn't matter who wins any specific individual match, only the tournament as a whole.
2. **Break it down.** In order to determine who won the tournament, I will need to determine who won each individual match. So I have to look at all of the matches and keep track of the number of wins for each player over the whole tournament, and compare the totals at the end to determine the result.
3. **Solve the problem by hand.** I grab a piece of paper and a pen and write out a quick solution to the problem given the input above. The result looks like this:

Input: "RP SR PS RS SR PP PR SR"

player 1's win count = ~~0~~ 2      player 2's win count = ~~0~~ ~~4~~ ~~5~~ 5

match #	1	2	3	4	5	6	7	8
result	RP	SR	PS	RS	SR	PP	PR	SR
winner	P2	P2	P2	P1	P2	Tie	P1	P2

Conclusion: Player 2 wins, 5 to 2

---

Try this for yourself. I used a table to organize the information, but you can keep track of it however you like.

4. **Write down the steps.** I come up with the following plan:

- (a) For each match in the tournament, determine the winner of the match and update that person's win count. If there was a tie, do not update anything.
- (b) At the end, output the name of the player with more wins, or output "Tie" if the counts are the same.

5. **Determine how to store the data.** I will store the matches in a list called `matches` (so that I can iterate through them) and have two variables called `player1` and `player2` to keep track of the win counts.

Now that I have planned the steps I will execute (the algorithm) and how I plan to store my data (the data structures I will need) I am ready to code the solution.

**Keep in mind:** You don't always need to follow exactly these steps every time you want to solve a problem. You can modify this approach however you like. As you solve more problems, you will determine what works best for you. However, it is always important to make some kind of plan! You should always have a good idea of your intended algorithm and data structures before you start to code.

## 8.3 HOW TO TEST AND DEBUG: THE BASICS

Once you have devised a strategy for solving a problem, it is time to start coding a solution. As you work to solve the problems in this section, you will probably run into a wide variety of errors. Some of these will be easily found and fixed, while others will be harder to track down. Fortunately, there are a few steps you can take to become better at **debugging** to find the errors in your programs.

### 8.3.1 READING ERROR MESSAGES

When Python gives you an error, it will always be shown in the form of a **traceback**:

```
File "count_vowels.py", line 25
    if i in vowels
        ^
SyntaxError: invalid syntax
```

The first line

```
File "count_vowels.py", line 25
```

indicates the file that caused the error, `count_vowels.py`, and the specific line number where the error occurred.

The next lines repeat the line where the error occurred and show a little "arrow" demonstrating where exactly the error occurred in the line

```
if i in vowels
    ^
```

Finally, the last line will give some additional information about the type of error that occurred. This error is a `SyntaxError`, a mistake in the structure of the code. This error was caused because this if statement is missing a colon (':') at the end of it:

```
SyntaxError: invalid syntax
```

There is another main class of error that you will also see: Semantic errors, which manifest as `ValueErrors`, `TypeError`s, `IndexErrors`, and several other types. These types help to give some information about the sort of error you are looking for in your code. We will discuss the difference between semantic and syntax errors below.

Unfortunately, the description only tells you that there is an error with the structure of the code. It doesn't tell you exactly what caused the error or what you need to do to fix it. It is up to you to look at your code and determine what caused the problem in that line.

### 8.3.2 SOMETIMES THE ERROR IS SOMEWHERE ELSE

Sometimes, the Python interpreter will report an error whose original cause is somewhere else. For example, this program has a simple error. We are missing a bracket at the end of the input statement on line 1:

```
string = input(
count = 0

for i in string:
    print(i)
```

However, when this program is run, the error message that prints looks like this:

```
File "count_string.py", line 4
    for i in string:
        ^
SyntaxError: invalid syntax
```

Python tells us there is an error in line 4, within the for loop declaration, rather than in line 1 where the error originated! This is because the Python interpreter is still looking for the closing bracket when it reaches the keyword `for`. Because it has not yet closed the `input` function call, it knows that the keyword `for` is incorrect.

The Python interpreter reports the error on line 4 for a reason, but this is not the true cause of the problem. Sometimes, you may need to trace your code and check for common errors, like missing parentheses, in the lines *before* the line where Python reports the error. You can do this by hand, or using a debugger. This is something to keep in mind as you continue to learn programming in Python.



---

### 8.3.3 SYNTAX ERRORS VS SEMANTIC ERRORS

If you see a syntax error, you have a problem with the **structure** of your Python code. You may be missing a bracket or semicolon, or have a variable name that is not allowed. You may also have a lexical error, in which you have used an incorrect token, but the Python interpreter reports these as syntax errors as well. When you have this type of error, you will see “SyntaxError” at the beginning of the error message.

A semantic error, on the other hand, indicates that you have a problem with the **meaning** of your code. Semantic errors will not be labelled as such; rather, they comprise most of the remaining errors you will see. For example, “ValueError” and “TypeError” are common semantic errors. For example, if we try to perform operations on variables which have incompatible types, Python will exit with a TypeError. This is what happens when we attempt to add a string to an integer, such as `"hello" + 2`.

Semantic errors can be harder to track down because often you have to carefully trace the meaning of your code in order to find the issue. This simply takes practice.

The main difference between syntax and semantic errors is that the program is checked for syntax errors before any lines of code are run. However, a semantic error may occur near the end of the program, after some or many lines have already been executed.

### 8.3.4 YOUR TASK: ADDITIONAL RESOURCES

Take a moment before you continue to read [this excellent article](#) about Errors and Exceptions in Python 3. Although it is well-worth reading from start to finish to supplement this section, here is a summary of several key error types taken from the end of the article:

- Tracebacks can look intimidating, but they give us a lot of useful information about what went wrong in our program, including where the error occurred and what type of error it was.
- An error having to do with the ‘grammar’ or syntax of the program is called a **SyntaxError**. If the issue has to do with how the code is indented, then it will be called an **IndentationError**.
- A **NameError** will occur if you use a variable that has not been defined, either because you meant to use quotes around a string, you forgot to define the variable, or you just made a typo.
- Containers like lists and strings will generate errors if you try to access items in them that do not exist. This type of error is called an **IndexError**.
- Trying to read a file that does not exist will give you an **FileNotFoundError**. Trying to read a file that is open for writing, or writing to a file that is open for reading, will give you an **IOError**.

You should also complete the section of the Waterloo Computing Science Circles on [Errors in Python](#). It will give you more practical experience understanding error messages and the types of errors you may encounter.

Resources:

- 
- You may also want to look at [Sections 8.1 and 8.2 of the official Python Tutorial](#) on Errors and Exceptions for more information.
  - If you are curious to learn more than what was covered here, read [this page](#). It is possible to **handle** exceptions in your program using `try-except` blocks and the `finally` statement. You can research these concepts on your own if you are curious.

### 8.3.5 DEBUGGING TIPS

Here are a few tips to help make debugging your code easier:

1. **Run your code often.** Don't try to write too much code all at once! If you try to solve the entire problem before ever running the code, you will make many mistakes. It is much harder to track down mistakes in a large code base! Instead, whenever you finish a section of code, run and test it. This will make errors much easier to track down because you know that they are **localized** in a small segment of your code.
2. **Add print statements to your code.** If your program returns no errors, but still does not return the correct result, there are two possibilities: either your algorithm is wrong, or you have made a mistake somewhere in your code. Since these types of mistakes do not cause exceptions that break your code, it is hard to track them down. Some examples of mistakes like this include:
  - Adding something that you should have subtracted.
  - An infinite loop that never terminates.
  - Forgetting a step in a process or algorithm.

Printing the state of your program can help you to track down the issues. For example, if you have an infinite loop, you can print the value of an iterator or other important variable to find out why it is not terminating. If you have multiple loops in your program, a print statement can help you find out which loop is causing the problem.

3. Use a **debugger** or other code-tracing program to walk through the Python statements step-by-step. A debugging program will often allow you to see the values of all variables and trace the execution of the program at each step, stopping wherever you like. In that sense, a debugger is like a super powerful print statement, allowing you to see exactly what is going on with your code.
  - A free, online debugger that works for Python and several other languages is [PythonTutor](#), [found here](#). Try tracing one of the programs you created in an earlier lab to learn how it works.
4. **Write comments throughout your code.** Do this to:
  - Plan out the steps of your algorithm in your code before you get started. Create a comment skeleton you can flesh out as you go along.
  - Allow a TA or coding partner to understand your code, work with you more effectively, and help you debug any issues more easily.

- 
- Help yourself understand your own code when you come back to it later. Comments are helpful not only for someone else, but also for yourself. This is especially true when you are working on an extremely large project.

## 8.4 PROBLEMS TO SOLVE

This section will provide you with a number of interesting problems that can be solved using only the skills taught so far. For each exercise, you will be given the problem description and some examples of input and output specifications. It is then up to you to solve the problem and output the correct result.

You should have all of the programming skills needed to solve these problems, although some of them may be easier after you have learned to create your own functions later in the course.

### In the Style of Morning Problems

Here are a few mechanical problems that test your problem-solving, programming, and basic math skills. These problems are phrased in a similar format to morning problems, and all have similar input and output specifications. However, a few of the problems have additional sections labelled “**Challenge**”, which provide an additional difficult exercise that would not be expected of you in a typical morning problem.

#### 8.4.1 FIND LARGEST INTEGER

**Description:** Write a program called `findMax.py` that finds the largest number in an unsorted (ie, random) list of integers. To avoid making this challenge completely trivial, you must implement it without using the `max` function.

**Input:** A space-separated list of integers.

**Output:** A single integer, the largest number in the list.

##### Sample Input 1

```
2 -43 3 34 12 53 27 -15
```

##### Sample Output 1

```
53
```

**Challenge:** Instead of reading the integers in from standard input, use the `random` module to randomly generate the list.

---

### 8.4.2 FIND THE LARGEST NUMBER ARRANGEMENT

**Description:** Write a program called `largest_num.py` that given a list of non negative integers, arranges them such that they form the largest possible number. For example, given `[5, 2, 1, 9]`, the largest formed number is 9521.

**Input:** A space-separated list of integers, all of which are guaranteed to be a single digit (0 - 9).

**Output:** A single number representing the largest integer that can be formed.

#### Sample Input 1

```
5 2 1 9
```

#### Sample Output 1

```
9521
```

**Challenge:** Solve the problem if the numbers do not need to be a single digit, but can be of any length. For example, here are a few example inputs and outputs:

#### Challenge Sample Input 1

```
5 50 56
```

#### Challenge Sample Output 1

```
56550
```

#### Challenge Sample Input 2

```
420 42 423
```

#### Challenge Sample Output 2

```
42423420
```

### 8.4.3 FIND THE SUM OF ALL NUMBERS IN A LIST

**Description:** In Python 3, there is a builtin function called `sum`, which adds all the elements in a list or container. For example:

```
>>> l = [1, 2, 3]
>>> sum(l)
6
```

However, it does not work for lists of the following type:

```
>>> l = ['a', 2, 3]
>>> sum(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Write a program called `new_sum.py` that will take a list of space-separated elements and add only the integer elements together. For our purposes, an integer element is one containing only digits and no other characters.

**Input:** A line of space-separated elements. Each element will always either be an integer (digits only) or a string containing at least one letter. The string could contain digits or symbols, but will always contain at least one letter.

**Output:** The sum of the list if only the integers are added.

#### Sample Input 1

```
a2@ 3 1 abc a
```

#### Sample Output 1

```
4
```

### 8.4.4 PIG LATIN

**Description:** Pig Latin is a game involving alterations to the English language. To create the Pig Latin form of an English word the initial consonant sound is transposed to the end of the word and an `ay` is affixed (Ex.: "banana" would yield anana-bay). More information on the rules can be found on the Wikipedia page [here](#).

Write a program called `to_pig_latin.py` that takes a string containing a regular English sentence and converts it to Pig Latin. Follow these rules:

For words that begin with consonant sounds, all letters before the initial vowel are placed at the end of the word sequence. Then, "ay" is added. For example, "smile" would become "ilesmay". For words that begin with vowel sounds, one just adds "ay" to the end.

**Input:** A string of English words, which will only contain letters of the English alphabet in either upper or lowercase.

**Output:** The same string converted to Pig Latin.

#### Sample Input 1

```
What glove are you wearing
```

#### Sample Output 1

---

atWhay oveglay areay ouyay earingway

## Additional Practice

A small collection of problems that have specifications that differ from a typical morning problem, but which are still good problems to solve.

### 8.4.5 SIMON

**Description:** In a file called `simon.py`, implement a text-based version of the classic game Simon (here is a link to the rules of the original version if you are unfamiliar).

In each round, your program will display a pattern (for example, a string of letters, digits, or symbols). The pattern will be visible for 1 second, and then it will disappear. The player must then copy the pattern correctly from memory in order to continue. Each round, the pattern will get longer, so it will become more difficult to remember all of the symbols.

Note that in order to make the game fun, you will need to clear the terminal at each step so the player cannot cheat by simply copying the pattern! You can do this using two lines from a builtin Python module called `os`:

```
import os
os.system('clear')
```

`os.system` is a function that will take the command given as its argument and execute it as if it were in the terminal. `'clear'` is a Bash command. Type it in the terminal to see what it does!

Note that you may also need to research the `random` and `time` modules in order to add new random elements to the pattern each round and to pause the game while the pattern is being displayed.

**Sample Game Run** This shows only the last round of the game. Your interface may look slightly different from this, as the exact input and output specifications are up to you.

```
Enter the pattern: EVNZLEFAD
Incorrect! The correct answer was: 'EVXZNLCLC'
Game over! You remembered the pattern correctly up to a length of 8.
```

**Challenge:** Make the game your own by adding a new rule of your choice. For example, perhaps the letters of the pattern now appear one by one on the screen before disappearing instead of all at once. The sky's the limit!

### 8.4.6 SOLVE THE NUMBER GUESSING GAME (HARD)

**Description:** In an exercise from an earlier lab, you were asked to write a simple number guessing program. Here was the description: “Generate a random number from 1 to 100 (or any range you

---

like). Prompt the user to enter a guess between 1 and 100. If their guess is wrong, tell them whether the guess is too high or too low, and ask them to guess again. The user should be allowed to keep guessing until they have successfully guessed the number.”

In this exercise, modify your game so that **you** set the secret number and the **program** must guess until it has found the number correctly. You should try to write a program that allows the computer to find the correct number **as quickly as possible**, with the lowest average search time. Your program should initially prompt you to enter a lower and upper bound (which restricts the guessing range) and should then start guessing. You can choose whether to automate the process (give the correct number to the computer and allow it to compare its guess with the answer automatically) or to tell the program yourself at each step whether the number is too high or too low. At the end of the program, have the computer print the number of iterations it took to find the value.

**Hint:** You want to implement an **efficient search algorithm** that your program will use to find the unknown number as quickly as possible. Note that guessing every number in the range sequentially, one by one, is not the most efficient method!

### Sample Program Run

```
Enter the secret number: 42
Lower bound: 1
Upper bound: 100

Guessing...
Found secret number 42 in 7 iterations
```

### 8.4.7 NUMBER NAMES (VERY HARD)

**Description:** Write a program called `number_names.py` that will take a 3-digit integer and convert it to the English spelling of the number it represents. For example, the integer 458 would be spelled “four hundred and fifty-eight”. For a refresher on the rules of spelling integers in English, [you can review this chart](#).

Note: You will need to find a way to isolate the individual digits of the integer and map them to the English equivalents. Consider: How do you know if a digit is in the ones, tens, hundreds, thousands, etc position?

**Note:** This problem is difficult to code correctly as there are many cases and conditions. If you are struggling, try to solve the problem for 2-digit numbers (10 to 99) first.

**Input:** An integer between 100 and 999 (inclusive).

**Output:** The spelling of the integer in English.

### Sample Inputs

```
273
401
```

---

750 100
------------

## Sample Outputs

two hundred and seventy-three four hundred and one seven hundred and fifty one hundred
---

**Challenge:** For an added challenge, modify your program so that it works for any integer from 0 to 9999 (so you must deal with numbers of varying lengths). If you can make that work, try numbers from 0 to 1,000,000! The solutions to these problems are difficult because they require managing many small details correctly. They are a pain to implement but will test your knowledge of programming.

## 8.5 LINKS TO EXTERNAL PROBLEMS ON KATTIS

If you are looking for other practice for your programming or morning problem skills, [Kattis](#) is a useful repository of programming problems. Kattis contains many problems that have similar types of input and output to morning problems you will see in class. The submission process is quite simple: make an account using your uAlberta email address and click on the orange cat in the upper left to see a list of problems. You can sort the problems by difficulty by clicking “DIFFICULTY” in the status bar at the top of the problem list.

For more information, feel free to follow the [tutorial here](#).

Here is a list (in rough order of difficulty) of some hand-picked problems from Kattis:

- [Hello World!](#): This is the very first problem to do: a problem that simply prints “Hello World!” to the terminal.
- [Stuck in a Time Loop](#): A wizard gives you a magic number, and you must count up to that number, starting at 1, saying “Abracadabra” each time.
- [Aaah!](#): Determine if Jon Marius can say “aaah” long enough to meet a doctor’s requirements.
- [Simon Says](#): Write a program that will play Simon Says perfectly.
- [Parking](#): Determine how far Michael will need to walk to visit all stores on his route, if he parks optimally.
- [Sibice](#): Help Mirko determine which matches fit in the box his mom gave him.
- [Railroad](#): Determine if it is possible to build a continuous railroad that does not have any dead ends. This problem requires more thinking than coding.
- [Planina](#): Determine the number of points on a square after N iterations of an algorithm. This problem requires more thinking than coding.



- 
- [Number Fun](#): For each test case, determine whether or not it is possible to produce the third number,  $c$ , using the first two numbers,  $a$  and  $b$ , using addition, subtraction, multiplication, or division.
  - [Encoded Message](#): Decrypt a secret message encrypted using a transposition cipher.
  - [Detailed Differences](#): Compare two strings and output a line denoting the differences between them.
  - [ABC](#): Read in three numbers as  $A$ ,  $B$ , and  $C$  and output them in a specified order.
  - [3D Printed Statues](#): You have a 3D printer that can either print a statue every day, or print another 3D printer. What is the minimum number of days you must take to print  $N$  statues? This problem requires more thinking than coding.
  - [Matrix Inverse](#): This problem involves some basic linear algebra knowledge, in order to compute the inverse of a  $2 \times 2$  matrix.
  - [Persistent](#): A challenging problem. Given a number, determine the smallest number such that computing the first step of its persistence results in the given number. To solve the problem, you will need to determine how to find the single-digit factors of a number.