

Computer Science Concepts & Programming

Overview

Hi! I'm Anna, a current sophomore studying computer science and information science at Cornell University. The first time I stumbled across computer science, I honestly didn't really know what it was. As I got to learn more about the subject, I realized that CS is so much more than computers, cubicles, and the huge blocks of code we see in movies – it means creating fashion technology, technologies to help disabled people and underserved communities, technologies to connect people around the globe like health professionals or to build communities that wouldn't have existed previously, and more. CS means having the power to create something. Over the course of this packet, I hope you will also find interest in computer science and the power it can wield in our lives! It might seem boring at first, but I promise that these foundational concepts help us develop more complex platforms and technologies.

This program will provide an introduction to fundamental programming skills in computer science (CS). This packet focuses on the philosophy of CS & the 4 CS core concepts. The packet has 6 lessons: each lesson will contain exercises to practice the skills learned, and a mini project at the end tying everything together. Answers to exercises will be at the very end.

Lesson 1 introduces what computer science is and some terms used in computer science. Lesson 2 discusses the logic and thinking behind computer science. Lessons 3-6 introduce what the 4 basic programming concepts are: variables, conditional statements, loops, and functions.

Lesson 1: Intro to CS

What is Computer Science?

CS is the study of computers/computer systems and how we can use them to create new technologies and solve problems. CS is important today as technology increasingly permeates our daily lives. More industries will need to rely on computer science knowledge to power their services or create new ones to adapt to these technological changes.

Computer scientists study a variety of subjects, including but not limited to:

- Algorithmic problem-solving
- Software & hardware design
- Data analysis (processing, visualizing & interpreting data)
- Human-computer interaction
- Programming (including game design)
- Security (i.e. cryptography)
- Web design
- Robotics

Careers & Applications of CS

Computer scientists learn how to solve problems through computational and critical thinking within the above subjects. The knowledge and skills they learn prepare them for careers in a variety of sectors:

- *Technology* – designing software and hardware systems, developing mobile or desktop applications, devices, and networks
- *Retail* – creating apps for customers to “virtually” try on items at home, analyzing consumer patterns to better curate products
- *Healthcare* – analyzing data to better predict diseases, developing security or privacy systems for medical records
- *Arts* – Composing digital music, designing special effects or improving CGI for movies
- *Weather forecasting* – developing models to predict hurricane behavior
- *Education* – interpreting data to better develop curriculum or learning methods, developing technologies to use inside the classroom
- *Finance* – designing automated trading services, strengthening bank security features

... and more! As you can guess, a majority of computing-related careers fall outside the technology industry. It’s also common for people with non-computing majors to later transition into a computing-related career.

Brief History of CS

Computers have come a long way – from room-sized machines to the tiny pocket phones we have today. Humans have used mechanical devices to aid calculations for hundreds of years. The abacus existed around 3000 B.C., used to count items by hand. In 87 B.C., the ancient Greeks created the Antikythera mechanism, the earliest known analog computer. This mechanism helped predict the motions of the stars and planets to help navigate the seas.

In 1703 A.D., German mathematician Gottfried Wilhelm Leibnitz introduced the *binary number system* for doing calculations, a system where information can be expressed by combinations of the digits 0 and 1. Ada Lovelace, an English mathematician, took computing another step further in 1843, when she wrote the first computer algorithm. Charles Babbage also invented a theory for the first programmable computer around this time.

The modern computing-machine era began with the Turing machine by Alan Turing and the invention of transistors at Bell Labs. These made modern-style computing possible. In 1976, Steve Wozniak built the circuit board for Apple-1, spurring home computing. Before, computing technology was exclusive to the government and later academic institutions. Tim Berners-Lee created the World Wide Web in 1990, and Marc Andreessen built the first user-friendly Web browser in 1993.

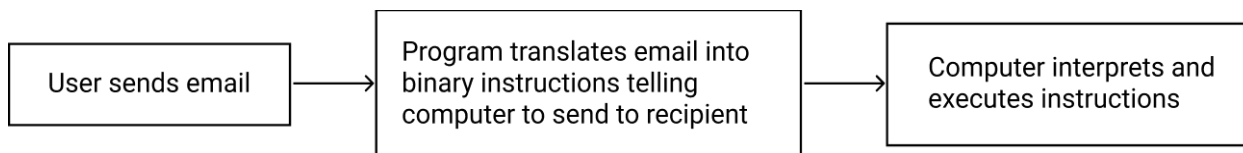
Computing technology took off – and since then, digital technology has advanced phenomenally, bringing the development of wearable technologies, microchips, neurotechnology, smart appliances, artificial intelligence applications, VR/AR (Virtual Reality/Augmented Reality) technology, and more!

What are Programs, Programmers & Programming Languages?

To define programming languages, we need to define what *programming* is. Programming is giving a set of instructions to a computer to execute. A *computer program*, then, is the actual set of instructions. You can think of a program as a recipe, with you being the computer and the recipe's author being the programmer! The recipe author (programmer) gives you a set of instructions (program) for you (computer) to follow.

Programming languages are the tools we use to write instructions for computers. Programming languages have grammatical rules (semantics and syntax), similar to an actual spoken language. Programming languages allow computers to communicate with each other, and for us as computer scientists to communicate with computers!

Computers “communicate” with each other in binary – strings of 0’s and 1’s. Programming languages translate binary into something we can read and understand. A real-world example is when you send an email: what happens behind the scenes is that a program translates your email into a set of binary instructions for the computer to interpret and execute and “send” to the recipient.



What does it mean for computers to communicate with one another using programming languages? Programming languages allow the parts of a computer (also known as the *hardware*) to communicate with one another. Hardware refers to the literal physical components of a computer – the microprocessor, RAM card, transistors, chips, the keyboard and screen, etc. Programming languages let these hardware components communicate and pass information between one another using binary.

Programming languages can typically be split into two categories: *low-level* programming languages, and *high-level* programming languages. Low-level languages are closer to machine code, or binary, and thus are generally difficult to use for humans (but certainly easier than binary). Machine code is tied to the computer hardware.



To help visualize this, we can use an example where we’re writing binary machine code to do an arithmetic operation. A programmer would need to know the binary codes identifying first the keyboard numbers, and then also the string of binary codes to send the numbers to the hardware that actually performs the calculation, and then even more binary codes to send the output back to the screen! Since low-level programming languages are close to machine code, they follow in that there are a lot of grammar and syntax rules to consider, making it difficult for programmers to use.¹

¹ Image from <https://marketbusinessnews.com/financial-glossary/machine-code/>

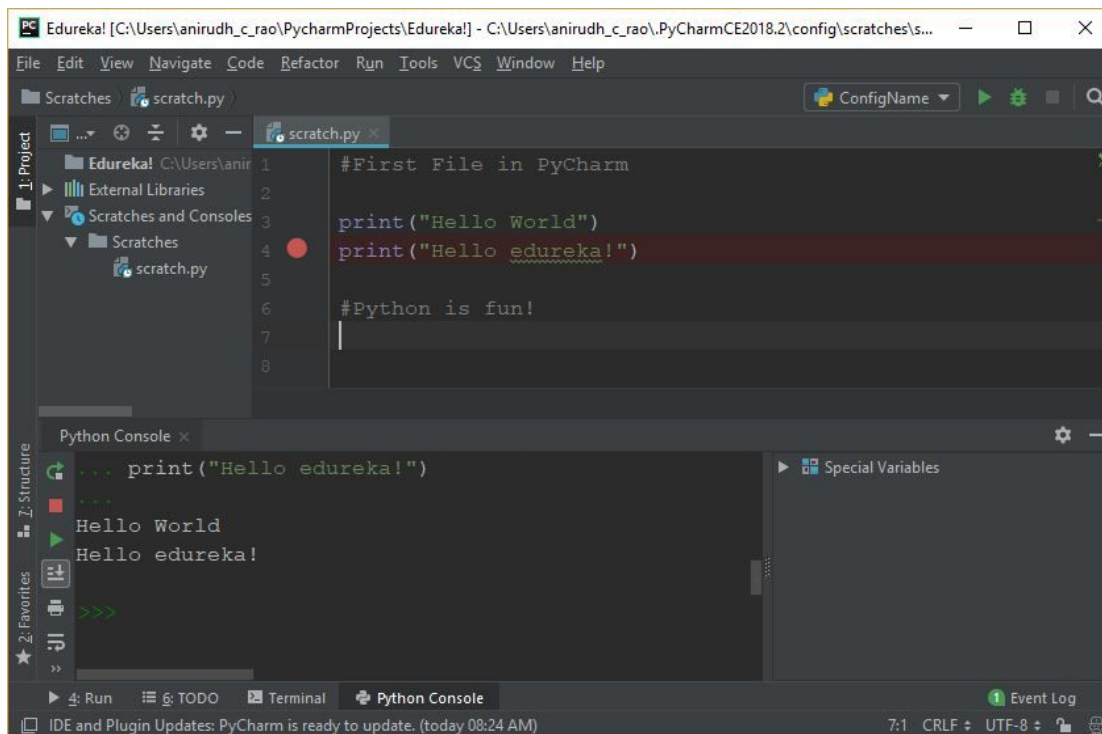
High-level programming languages *abstract*, or simplify, the computer hardware away and allow programmers to write computer programs (instructions) without having to worry about the details of the computer. High-level languages use words that we use in our everyday lives, making it easier for us to read and write complex programs! However, high-level languages take longer to translate into machine code for the computer. With the advancement of computers nowadays, though, the difference between low- and high-level languages is barely milliseconds!

Usually, when people talk about programming languages, they're referring to high-level programming languages. Some examples of such high-level languages include: Python, Java, C++, JavaScript, Ruby, PHP, and countless more. Each language contains different syntax, grammar, and semantics. The two most common are Python and Java. In this packet, we will be using Python. Python is a versatile language used for a variety of purposes and fields, including but not limited to data science, web development, and machine learning.

Why do we have so many programming languages? Because different languages have different rules, we can use the ones that suit our needs best. For example, a common basic web development language is HTML. However, HTML websites are *static* – meaning that each webpage is delivered, or looks, exactly the same to every user. But what if we wanted our website to have a personalized title, such as “Hello [NAME]” for each user? A programmer might choose to use React in this case to build a more dynamic and personalized website. There's no “best” programming language – it's all up to opinion and needs!

Integrated Development Environments (IDEs)

How can we create such programs on a computer?



2

An Integrated Development Environment (IDE) is a piece of software that allows us to write and develop programs using code. You can think of it like a canvas for paintings – you need a canvas to create paintings on, just like you need an IDE to write code and create programs on. There are many different IDEs to choose from depending on the type of program you want to write.

² Image from <https://www.edureka.co/blog/pycharm-tutorial>

Usually, in an IDE, there are 2 main spaces: the *code editor* and the *terminal*. The code editor is where you write your code. In the image above, the code editor is the main area in the middle. On the left bar, you can see all your folders and program files. Below the editor is the terminal – a place for you to actually see any output from your code! The terminal is also called the *console*, where again any output of your code will appear.

Above, the programmer is writing code in a file called `scratch.py`. There are currently two instructions in this program, on lines 3 and 4 respectively:

```
1. print("Hello World")
2. print("Hello edureka!")
```

Let's break these two lines of code down:

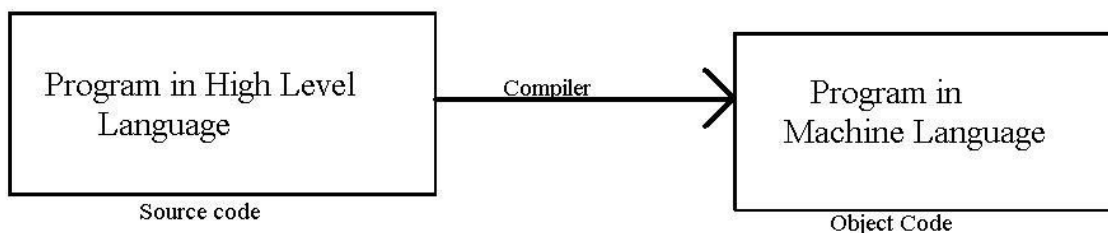
1. Print (output) "Hello World" to the console
2. Print "Hello edureka!" to the console

A `print` statement in Python tells the computer to print whatever is inside the parentheses out to the console. `print` statements can be helpful for programmers when *debugging* code, or fixing errors in the code. We will see this in the next few lessons!

In order for the code to actually execute, the programmer needs to *run* the code. You can think of running code like packaging up a box before handing it off to the computer to actually execute! To run code, all you need to do is press the run button. The run button is the triangle, seen in the image on the top right corner and on the left sidebar of the console. In the image, the programmer has already run the code. As a result, in the console you can see the output!

In this packet, we won't be needing IDEs, but they're important to know about! IDEs provide a workspace for us to create programs using *programming languages*.

How IDEs Work



3

In order for our computer to actually execute our program, our code actually goes through a process in order to translate it into executable instructions for the computer. In this process, we call the high-level code we write in programs the *source code*. When you hit the run button on an IDE, what happens is that the source code passes through

³ Image from <https://medium.com/coding-den/the-compilation-process-a1307824d40e>

a *compiler*. A compiler is a program that processes statements written in a particular programming language and translates them into machine code for the computer to process. During this step, the compiler also checks for any errors in the code, and prevents the program from running if any errors are detected. The compiler *parses* (analyzes) all the code statements in the program and builds the output machine code. We typically refer to this output code as *object code*. The object code contains executable instructions that the computer can read and execute. This whole process is called *compilation*.

Vocabulary Review

- **Computer science:** The study of computers/computer systems and how we can use them to create new technologies and solve problems.
- **Binary system:** A system where information can be expressed by combinations of the digits 0 and 1, used in machine code to give instructions to computers
- **Programming:** Giving a set of instructions to a computer to execute
- **Computer program:** A set of instructions that a computer can execute
- **Programming language:** The tools we use to write instructions for computers. They have grammatical rules (semantics and syntax), similar to an actual spoken language
 - **Low-level programming language:** Closer to machine code, or binary. Often hard to use.
 - **High-level programming language:** Abstract machine code away and use words that we use. Easier to use.
- **Integrated Development Environment (IDE):** A piece of software that allows us to write and develop programs using code
 - **Code editor:** The space in an IDE where programmers can write and edit code.
 - **Terminal/console:** A space where any outputs or *bugs* (errors) from your code will appear after you run your code.
- **`print("Hi there!")`:** Prints "Hi there!" to an IDE's console/terminal.
- **Source code:** The high-level code programmers write in programs.
- **Compiler:** A program that processes statements written in a particular programming language and translates them into machine code for the computer to process
- **Machine/object code:** The output of a compilation process

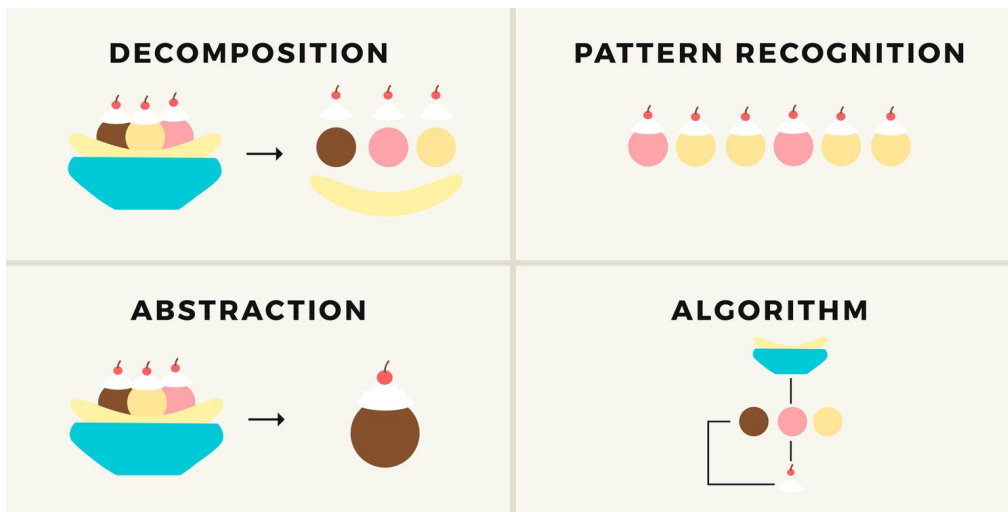
Exercises

- 1.1) What are some other applications/examples of CS in industry or everyday life that you can think of?
- 1.2) In our brief history of CS, we mentioned the Turing machine created by Alan Turing. What do you think this is? Can you think of any other computer scientists you know?
- 1.3) What is the difference between low- and high-level programming languages?
- 1.4) Describe what happens when you run a piece of code.
- 1.5) You may have also heard of the terms *information technology (IT)* and *information science*. What do you think is the difference between IT, information science, and computer science?
- 1.6) Write the code that would print "Hello [YOUR NAME]" to a Python console!

Lesson 2: Computational Thinking

According to the BBC, computational thinking is a thought process that allows us to “take a complex problem, understand the problem, and devise possible solutions. We can present these solutions in a way that computers and humans can both understand.” Computational thinking helps us tackle problems using 4 main technique:

1. **Decomposition:** Breaking down problems into smaller, more manageable problem chunks
2. **Pattern recognition:** Looking for similarities among problems
3. **Abstraction:** Identifying important information while ignoring irrelevant details
4. **Algorithms:** Developing a step-by-step solution to a problem



We can create computer programs that solve complex CS problems using computation thinking: we take a complex problem and break it down into chunks (decomposition). Each of these smaller problems can be looked at individually using knowledge of similar problems that have been solved before (pattern recognition), focusing on the important information while ignoring unimportant details (abstraction). By doing this, we can create step-by-step rules or instructions for each of these small problems (algorithm). Put all together and we have a program that solves the initial complex problem! Essentially, computational thinking helps us figure out what to tell the computer to do.

It's important to note that computational thinking isn't only limited to computer science. Computational thinking can be used across a variety of fields, and refers to a way of thinking and solving problems rather than just a field itself. For example, students can use computational thinking to perform literary analysis on books like *Hamlet* or *Lord of the Flies*. They can break down (decompose) the themes and ideas of the literature to better understand the work and the author's message.

Decomposition

Decomposition means breaking down a complex problem into smaller, more manageable parts. We can focus on the individual parts to solve the whole. Decomposition is important because without it, problems can be hard to solve. Imagine trying to explain how to build a bicycle without knowing the parts and pieces of it. It would be incredibly difficult! Decomposition means explaining how to build a bicycle by breaking it down into processes and parts.

⁴ Image from <https://www.tinythinkers.org/benefits>

A real-world example of decomposition could be planning a birthday party. How can we break down this bigger task into smaller, more actionable to-do lists? Here are some smaller problems that we could break this task into:

- Invitations
- A birthday theme
- A venue/location
- The total budget of the party

From these parts, we can further break the task down into smaller problems:

<ul style="list-style-type: none"> ● Invitations <ul style="list-style-type: none"> ○ Choosing who is invited ○ Creating invitations ○ Sending invitations ○ Gathering a final list of who can come 	<ul style="list-style-type: none"> ● A birthday theme <ul style="list-style-type: none"> ○ Choosing a birthday theme ○ Buying decorations ○ Buying a cake
<ul style="list-style-type: none"> ● A venue/location <ul style="list-style-type: none"> ○ Finding & contacting potential locations ○ Confirming the location & time ○ Paying for the location 	<ul style="list-style-type: none"> ● A total budget <ul style="list-style-type: none"> ○ Splitting the budget into parts

And etcetera! There are a lot of ways you could decompose this problem into smaller problems. Let's practice how to do this with something computer science related:

Exercises 2.1) Imagine you want to create your first app. How would you decompose this task? There are no wrong answers here, and the goal is to understand how you can decompose a complex problem into manageable parts!

Pattern Recognition

Pattern recognition means looking for similarities or "patterns" between the smaller problems that we have decomposed. Patterns can help us solve problems more quickly and efficiently. For example, imagine that we're writing an essay. We can decompose this problem of writing an essay into 5 smaller "problems": the introductory paragraph, 3 body paragraphs, and a conclusion paragraph. Here, we already have a pattern – the 3 repeating body paragraphs! Additionally, there exist patterns within each body paragraph: we know that each body paragraph will most likely be similar in structure, consisting of a topic sentence, evidence, and analysis. So, we see the pattern that each body paragraph will need a topic sentence, a piece of evidence, and some analysis. Patterns allow us to create formulas that we can refer back to everytime we need it!

$B\text{-paragraph} = t + e + a$ (With t being the topic sentence, e being the evidence, and a being the analysis. So, instead of rewriting the steps of a body paragraph over and over again, we can simply write " $B\text{-paragraph}$ " and refer back to our formula.)

Exercise 2.2) One of the ways that pattern recognition proves vital is in cryptography, the practice of writing or solving codes. One famous encryption (i.e. encoding information) technique is the Caesar Cipher, a substitution cipher where

each letter in the alphabet is replaced by a letter that is some fixed number of positions down the alphabet. For example, for a shift of 2: A = C, B = D, ..., Y = A, and Z = B. People write messages using this encrypted alphabet, and pass along the secret shift to the recipient. However, this cipher isn't the most secure – it can be cracked (decrypted) through pattern recognition. Use pattern recognition to decipher the message in the last sentence. What is the shift? *Hint: which letters appear the most in English words?* FTQ OMQEMD OUBTQD IME ZMYQP MRFQD VGXUGE OMQEMD

Exercise 2.3) Take a look at the following string of numbers: 0 1 1 2 3 5 8 13 21... This pattern is called the Fibonacci Sequence, and is a famous formula in mathematics. What is the pattern in the sequence? (*Hint: It has to do with addition!*). Can you figure out a “formula” for the pattern, like in the B-paragraph example given earlier?

Exercise 2.4a) Let's practice decomposition and pattern recognition with another real-life example. Imagine that you're making a peanut butter and jelly sandwich. First, decompose this problem into smaller problems (steps) that you can take.

Exercise 2.4b) Now using your decomposition, can you recognize any patterns/repeats in your smaller problems? For all the patterns you recognize, try to create a reusable “formula” that simplifies the patterns/repeats!

Exercise 2.5) What might happen if we don't look for patterns?

Abstraction

Abstraction refers to ignoring, or filtering out the details of our problem that we don't need to concentrate on. After we decompose our problem into smaller problems and recognize patterns among the sub-problems, abstraction helps us identify general characteristics among the patterns and filter out the irrelevant details in order to create the most efficient solution. We can then create a model or representation of what we're trying to solve!

We actually used abstraction in our previous example with the essay. We noted that all body paragraphs have general characteristics, namely the topic sentence, evidence, and analysis. In addition, each body paragraph has specific characteristics/details that we didn't focus on, such as the source of the evidence, how long the analysis is, what the topic sentence is about, etc. In order to write any general essay, we need to know the general characteristics of the body paragraphs, but not the specifics! Here we abstract away the specific details to focus on the overall structure of the essay. Once we know what a basic essay structure is like, then we start fleshing out the details of our essay.

Another great example of abstraction is with literature – when you read a book, what are the main ideas or themes? These main ideas are separate from the details of the book. Let's consider all books in general: what are the general patterns across all books? What are the specific details?

General Patterns	Specific Details
We need to know the main ideas/themes	We don't need to know what or how many themes there are.
We need to know that a book has characters	We don't need to know the number of characters or who the characters are
We need to know that a book has chapter	We don't need to know the number of chapters or what the chapter titles are

Exercise 2.6) What might happen if we don't use abstraction?

Exercise 2.7) Say we wanted to create a random dog generator. What are the general characteristics (patterns) of most dogs? What are the specific details of dogs that we don't need to focus on?

Algorithms

Algorithms are step-by-step instructions to solve a problem. An algorithm is *not* the solution, but rather a problem-solving method – the algorithm is the set of instructions given to a computer to execute, but the actual solution is what comes out of the algorithm (what is changed). In an algorithm, it's important to succinctly describe each instruction, and plan out the order in which to execute them. A computer only can do what is given to it, so writing good algorithms is important to tell the computer exactly what we want it to do!

Algorithms exist all around us. If you've ever brushed your teeth, cooked a recipe, or followed an instruction manual, then you've executed an algorithm! For example, we can think of a cake recipe as a high-level algorithm: we are provided instructions that we must follow step-by-step in order to get to our solution, the baked cake.

Millions of more computer science related algorithms exist in common services such as Google maps (calculating fastest and/or shortest route), Netflix or any social media (curating feed to fit your interests), Tesla (self-driving cars), patient medical data in hospitals (encrypting the data to ensure cybersecurity), and more!

Exercise 2.8) Write an "algorithm" describing your morning routine. For example, maybe the first two instructions could be:

1. Waking up.
2. Stretching for 5 minutes.

How can you use pattern recognition and abstraction to simplify this algorithm?

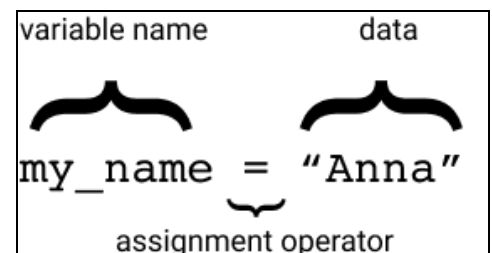
Lesson 3: Variables

Recap #1: Python is a high-level programming language that we can use to write instructions for computers in program files. Computers take these program files and translate them into machine code, then execute the instructions.

Recap #2: What does the statement `print("Hello world")` do? This statement tells the computer to print the phrase "Hello world" to the console when we run the code in an IDE. If we changed the phrase inside the quotes, it would print out the changed phrase!

Exercise 3.1) Write the code that prints out your favorite quote to the console.

What are variables? In CS, *Variables* are used to store pieces of data that we want to reuse. For example, if there is an email we want to remember, a greeting we want to present, or an interest rate we want to reuse, we could create variables storing these values. There are 3 parts of a variable in Python: the *name*, *assignment operator*, and *data*.



We *assign* data to a variable using the equals sign (=). In the example, we store the text "Anna" in a variable called `my_name`. Now everytime we refer to `my_name`, we are actually referring to "Anna".

There are limitations to how we can create variables: 1) They cannot have spaces or symbols in their name other than an underscore (`_`). 2) They can't begin with numbers but can contain numbers after the first letter (i.e. `name_version_5`)

What if we wanted to store pieces of data other than text? Python has several *data types* that we can assign to variables:

1. **Strings:** These are the blocks of text we have been using thus far, like "Anna" or "Hello world". Strings can be encased in double quotes ("Anna") or single quotes ('Anna'). So `my_name = "Anna"` is equivalent to `my_name = 'Anna'`.
2. **Numbers:** Python has multiple numeric data types.
 - a. An **integer** or `int`, is a whole number. It has no decimal point and contains all whole negative and positive numbers (... , -2, -1, 0, 1, 2,...). Integers could be used when counting the number of people enrolled in a class, the number of doctors available at the moment, the number of fabric colors there are in a factory, etc.
 - b. A **floating point number** or `float`, is a decimal number (i.e 3.7, 100.00, etc). It can be used to represent fractions or precise measurements. Floating point numbers could be used when calculating the average grade of students in a class, the amount of money available in a bank account, the amount of a certain chemical in a medication, etc.

Let's take a look at a piece of code:

```
greeting = "Good morning!"  
# Prints "Good morning!"  
print(greeting)
```

What's happening here? First, we create a variable named `greeting` that stores the string "Good morning!". Then, we write a *comment*. Comments are ignored by the program and don't do anything outside of providing context.

In Python, we write comments starting with the `#` symbol. Python understands that anything after the `#` symbol is a comment. We write comments for many reasons, commonly to 1) Provide context for a piece of code, 2) Help other people reading the code understand what it means/does, 3) Ignore pieces of code that we don't want to run.

Exercise 3.2) Which out of the three reasons does the below comments serve?

A) # Calculates the number of words in a song
`song_calculation()`

B) # Stores the number of states in the US
`num_states = 50`

Exercise 3.3) Create a variable name `my_age` and set it to your current age. Then, print out the value of `my_age`.

Using number data types, we can do addition, subtraction, division, and multiplication in Python using `+`, `-`, `/`, and `*` respectively. For example: Notice that when we do division, Python converts all `ints` into `floats`. Additionally, notice how we can use variables to store numbers and do calculations with them! In the last line, we refer to the number 7 and 4 as *literals* because they're actual numbers that we use and not variables storing the numbers.

```
# Prints "99"  
print(50 * 2 - 1)  
  
# Prints "8.0"  
print(64 / 8)  
  
lucky_num = 3  
  
# Prints "33"  
print((7 + 4) * lucky_num)
```

Exercise 3.5) Similarly, take a look at the following statements:

```
my_name = "Anna"
```

```
print("Good morning" + my_name)
```

What prints to the console? *Hint: Be careful of spaces!!*

In Exercise 3.5, we are doing something called *string concatenation*! String concatenation means taking two or more strings and combining them into one. Using the + operator, we can “add” strings together to create a new string!

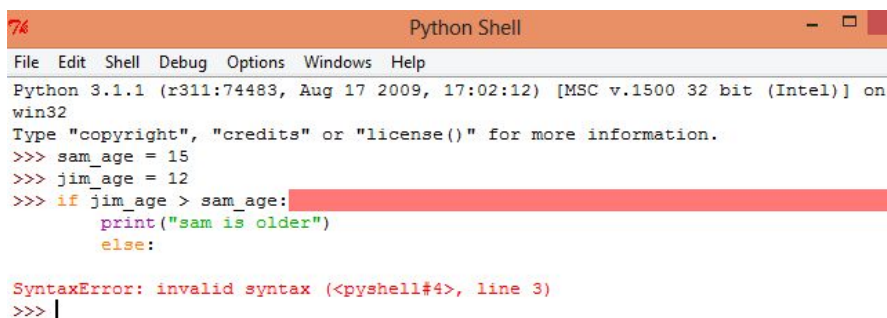
Exercise 3.6) Store your name in a variable called `my_name`. Then, store your favorite animal in a variable called `favorite_animal`. Print out the statement “My name is [NAME] and my favorite animal is [ANIMAL]”.

We can also concatenate number data types with strings, but we would first need to convert the number into a string using `str(num)`. However, if we want to print a number, we can *pass* it as a value rather than using `str()`. For example:

```
dog_age_string = "My dog's age is "  
dog_age = 3  
dog_age_string_2 = " years old."  
  
# Stores "My dog's age is 3 years old."  
dog_age_full_string = dog_age_string + str(dog_age) + dog_age_string_2  
  
# Prints "My dog's age is 3 years old."  
print(dog_age_string, dog_age, dog_age_string_2)
```

Exercise 3.7) Store this month in a variable with a name of your choice, and today’s date in another variable with a name of your choice. When coding, it’s recommended to always name variables in a descriptive way that gives clues or context into what they are supposed to store. Then, print out the string “Today is [MONTH] [DATE] .”

When we do all sorts of calculations and concatenations, as humans, it’s natural to accidentally make mistakes. In programming, we call these mistakes *bugs* (errors), coined by Grace Hopper, an American computer scientist and US Navy rear admiral who was a pioneer of computer programming. Bugs pop up when we run our program – during the compilation program, the compiler detects mistakes and *throws* an error, preventing the program from executing. On an IDE, this normally shows up as an error message (usually in the console).



The image shows a screenshot of a Python Shell window. The title bar says "Python Shell". The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The text area shows the following code:

```
Python 3.1.1 (r311:74483, Aug 17 2009, 17:02:12) [MSC v.1500 32 bit (Intel)] on win32  
Type "copyright", "credits" or "license()" for more information.  
>>> sam_age = 15  
>>> jim_age = 12  
>>> if jim_age > sam_age:  
    print("sam is older")  
    else:
```


The line `else:` is highlighted in red. Below the code, an error message is displayed:

```
SyntaxError: invalid syntax (<pyshell#4>, line 3)  
>>> |
```

⁵ In this image, the error message shows up at the bottom starting with the line “SyntaxError: invalid syntax”. Sometimes, error messages also give information about where the error occurred in the code so you can go back and try to locate the error, as also seen in the image.

⁵ Image from <https://stackoverflow.com/questions/19214053/python-syntax-error-invalid-syntax>

There are many *types* of errors that could occur. Some include:

1. `SyntaxError`: Something is wrong with the syntax of your program, such as wrong or missing punctuation, missing parenthesis, or commands in the wrong place.
2. `NameError`: Python detects a word that it doesn't recognize.
3. `ZeroDivisionError`: It's in the name! Python throws this error if you try to divide by 0.

Exercise 3.8) The following pieces of code contain 3 errors. What type of errors are they? Debug the code so that no errors occur when we run it.

```
weather_today = "Today was a beautiful, clear sunny day."
print((3 + 6) + ((9 * 2) + 1)
weather_tomorrow = "Tomorrow will be more rainy."
print(weather_tomorrow + I like rainy weather too.)
```

For number data types, there are 2 more operators we can use: the exponential (**) operator and modulo (%) operator! The exponential operator does what it says in the name: it performs exponential calculations like 7^3 , 2^{10} , etc. Here, we are representing exponents using the carrot (^), but Python uses a double start (**). The modulo operator (%) gives the remainder of a division. If the number is divisible, then the remainder is 0. We use modulo in programming when we want to do something every nth time.

Exponent (**)	Modulo (%)
# Prints 7 * 7, or 49 print(7 ** 2) # Prints square root of 225, or 15 print(225 ** (0.5))	# Prints remainder of 33/3, or 0 print(33 % 3) # Prints remainder of 42/8, or 2 print(42 % 5) # Prints remainder of 1/3, or 1 print(1 % 3)

Exercise 3.9) How can we check if the square of 11 is even using the exponential and modulo operators? Write a print statement that checks this.

Exercise 3.10) Imagine that there are 83 students in the class. The professor wants to split the class into groups of 5. How many students will be left over in the last group? Print out this number.

```
steak_story = """
Hello there!
Today, I learned how to
cook steak.
My first try wasn't that
great.
I burned all the steak.
"""
```

What if we wanted to store a really long string that spans multiple lines? If we tried doing this, we would get a `SyntaxError`. Instead, we can use triple quotes (""" or ''') to indicate *multi-line strings*. Sometimes, we may also want to use a quotation in our strings. In order to use a quotation mark within a string, we need to use an *escape character*: a backslash followed by a quotation (\ " or \ '). This tells Python that the quotation doesn't end the string, but rather is a part of it.

While variables store values, they can be updated as needed! We can reuse variables to represent a changing state. For example, maybe we are writing a program that allows you to select a medical patient from a database of patients. In this program, we may want to store the patient's ID or name in a variable, and update the variable everytime we select a

new patient. Another example could be calculating the averages of a class: we store this average, say 80, in a variable called `class_avg = 80`. everytime a new assignment is graded, we want to recalculate the average and update `class_avg` to reflect the new average.

We can update variables by simply reassigning them to the new value, similar to *initializing* (creating) them! For example: `class_avg = 88` or `class_name = "New Name"`. We can also update variables using the *plus-equals* (`+=`) operator. This is a shorthand for updating variables by taking the current value of the variable and adding the new value to it. This can be helpful when we're trying to take the sum or total of something, or add onto a string.

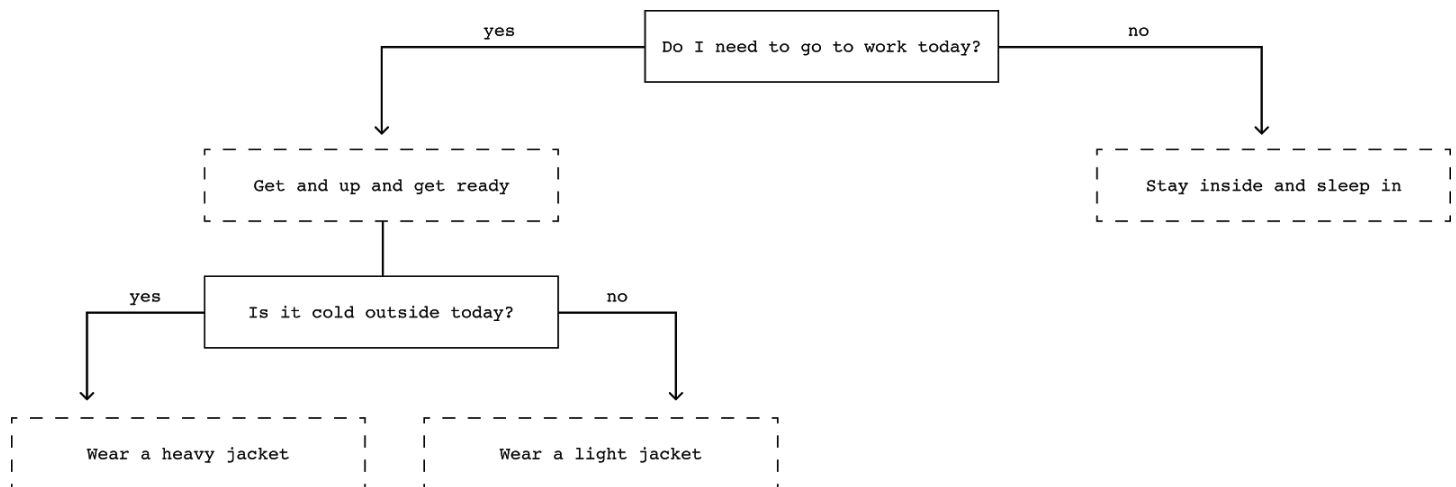
```
# Prints out "New Name TBD"
print(class_name += " TBD")
```

Exercise 3.11) Add onto the steak story variable to turn it into a successful story! Then, pick a random number and store it in a variable named accordingly. Update your random number by 3. Then print out the steak story along with the line "It took me [YOUR RANDOM NUMBER] tries."

Lesson 4: Conditionals

Conditions exist all around us in our everyday decisions: Do I have work today? If so, I'll get up and start getting ready. If not, I'll stay inside and sleep in for a little longer. Suppose I have work today (darn it). I have to get up and start getting ready for my day. Is it cold outside today? If so, I'll wear a heavy jacket, but if not, I'll wear a light one.

These questions and decisions describe a *control flow*, where we can take different *paths* depending on what we are conditioning on. For example, in the first question "Do I have work today?" we are conditioning on if we have school or not. We can take two different pathways depending on the answer, a yes or no.



Computers are similar! They can take different paths of instructions, going from top down, depending on different conditions. Whenever a condition is met, the computer will take that path of instruction.

Exercise 4.1) Come up with a conditional statement of your own. Define two paths you could take depending on whether this condition is *true* (yes) or *false* (no).

Booleans

Recap: Variables can take on different data types, such as `string`, `int`, and `float`. The *Boolean data type*, or `bool` type, is another data type which can take on only two values: `True` or `False` (Note: capitalized letters). We can create `bool` variables the easiest way by assigning them to either a `True` value or a `False` value: `is_cold_today = True` or `is_cold_today = False`. These two values represent the two paths that we can take depending on if something is true or false, like in the diagram!

Using booleans, we can create *boolean/conditional expressions*, or statements that evaluate to either true or false. Going back to our original example, the question “Do I need to go to work today?” can be transformed into a boolean expression: `I need to go to work today`. This expression can evaluate to either `True` or `False`.

Exercise 4.2) Come up with 3 more boolean expressions not from the example. Feel free to create a flow chart diagram like in the previous page!

Exercise 4.3) What kinds of statements wouldn’t qualify as boolean expressions? Name an example.

Relational Operators

We can create boolean expressions using *relational operators*, which compare two things and return either `True` or `False` depending on the comparison! Relational operators are also referred to as *comparators* for this reason. There are 6 types of comparators:

- Equals: `==`
- Not equals: `!=`
- Greater than: `>`
- Less than: `<`
- Greater than or equal to: `>=`
- Less than or equal to: `<=`

The exclamation mark (!) represents “not” in comparators, which we’ll explain in detail later. Let’s take a look at some examples of boolean expressions using these comparators:

```
# Prints "False"
print(5 > 10)
# Prints "False"
print(6 * 3 != 18)
# Prints "True"
print(6 + 4 * 2 <= 14)
# Prints "True"
print(2 ** 2 == 4)
```

Exercise 4.4) What do these boolean expressions evaluate to?

- `print(1 + 1 != 2)`
- `print(254 % 5 <= 10)`
- `print(254 / 5 > 10)`

Exercise 4.5) Come up with 3 of your own boolean expressions!

We can also compare strings using relational operators, namely *equals* (`=`) and *not equals* (`!=`). However, keep in mind that comparing strings is *case sensitive*. That means that `print('a' == 'A')` evaluates to `False`!

We can also use the other relational operators to compare strings; however, in this case the compiler will compare each letter to one another one-by-one. Depending on the character table of the computer, the evaluation may be different. This is because every character has a numerical value associated with it according to the computer. Remember when we talked about how computers communicate in binary (system of 0's and 1's)? Associating each letter with a number helps computers translate that associated number into binary code.

We can also assign boolean expressions to variables! Like how we assigned `is_cold_today = True` or `is_cold_today = False`, we can replace the True or False assignment with equivalent boolean expressions: `is_cold_today = 5 < 10` or `is_cold_today = 6 + 3 != 18`

Exercise 4.6) Create a variable named `test_bool`. Assign `test_bool` to `"true"` (with quotations). What data type does `test_bool` represent?

Exercise 4.7) We can check the types of variables by using `type(variable)` and printing it out to the console in an IDE. For example, to check the type of `test_bool`, we would use: `print(type(test_bool))`. Change `test_bool` to a bool type by assigning it to a boolean expression.

Boolean Operators

What if we wanted to check more than one boolean expression at a time? For example:

- Is it cold outside *and* will it rain later today?
- Do I want cereal for breakfast *or* do I want an apple?
- Did I close the window blinds *and* did I turn off the lights before going to bed?

We can create larger boolean expressions using *boolean operators*, or *logical operators*. There are 3 main boolean operators:

- **and**: If it is morning time *and* the sun is out, then...
- **or**: If it is morning *or* the sun is out, then...
- **not**: If it *isn't* morning, then...

If you've heard of logic tables, then you might know that each boolean operator has a *logic table* associated with it. A logic table determines what the resulting value is (True or False) of some number of expressions. The logic tables for and, or, and not are as follows:

A	B	A AND B	A OR B	NOT A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Where A and B represent boolean expressions (conditions). Notice that for `and`, if either conditions evaluate to `False`, then the whole boolean expression (`A AND B`) evaluates to `False`. *Both* conditions must be true in order for the whole boolean expression to be `True`. For `or`, if either conditions evaluate to `True`, then the whole expression is true! For `not`, not simply means flipping the value.

Exercise 4.8) Write a boolean expression (can be like the morning example!) for each of the boolean operators. Feel free to also write the pathways that occur from each evaluation (`True` or `False`) :)

Exercise 4.9) What do the following boolean expressions evaluate to?

- a) `(3 * 4 == 12) and (9 % 5 > 5)`
- b) `(10 - 12 >= 0) or (8 ** 2 < 100)`
- c) `not (12 + 16 > 25) and (5 * 3 == 15)`
- d) `not (8 + 33 <= 40) or (not (9 + 10 != 19))`
- e) `(not (26 % 5 == 3 and 82 - 7 + 5 > 77)) or ((not (12 + 41 == 53)) or 76 % 7 == 1)`

If/Else Statements

Let's go back to morning example we used when describing logical operators:

- `and`: If it is morning time *and* the sun is out, then...
- `or`: If it is morning *or* the sun is out, then...
- `not`: If it *isn't* morning, then...

If/Else statements represent two different pathways based on a condition (our boolean expression). Remember the diagram at the beginning of the lesson? Each box had two branches stemming from either a `True/False` decision – in programming, the “yes” branch represents an *if statement*, and the “no” branch represents an *else statement*. For example: If it's morning time, then I'll go outside for a walk. Else, I'll stay inside and watch a movie.

Can you pick out the boolean expression in the example? Right, it's “if it's morning time”, and we are checking to see if this condition is true. If it is, then we execute the rest of the if statement and go outside for a walk! If the condition is *not* met, we execute the else statement and stay inside. Some real world situations where we might use if/else statements includes online retail (if our total amount is over \$50, then we get free shipping), education (if we have completed at least 5 lessons, then we receive a virtual badge/reward), wearable technology (if a button is activated then it will call 911), and more!

In Python, we write if/else statements in the following format on the right: Don't worry about what `go_walk()` and `stay_inside()` mean. For now, they represent our two paths, but we'll go in depth about what the syntax means later. Else statements always go hand-in-hand with if statements. They describe the other pathway, if any, to an if statement.

```
if is_morning:
    go_walk()
else:
    stay_inside()
```

Exercise 4.10) Let's write an if/else statement in Python for our online retail example: if our total amount is over \$50, then we get free shipping. Otherwise, shipping is \$5. First, create a variable that stores the total amount and initialize (set) it to a number of your choice. If the total is over \$50, then print out “You get free shipping!” If the total is less, then print out a converse statement.

Exercise 4.11) Imagine that you're taking a tough class. In order to pass the class, your grade must be equal to or greater than a 70.0, *or* your final exam score must be greater than or equal to 85.0. Otherwise, you must retake the class. Write an if/else statement in Python that describes this scenario.

Exercise 4.12) Think of 2 other scenarios where you would use an if/else statement in real life. For example, if you have enough batter, then you'll make a big cake. Else, you'll make small muffins.

Else If Statements

We've talked about if/else statements. But there's one more I've been hiding from you guys: `elif` statements.

Else if statements provide an *alternative path* to an `if` statement if there's more than two paths you want to define. Elif statements check another condition if the `if` statement is `False`. The computer runs through `if/elif/else` statements in order, checking each condition from top to bottom. `Elif` statements go hand-in-hand with `if` statements, much like `else` statements. It wouldn't make sense to have an alternative path without the initial path!

```
if total >= 75:
    print("Free shipping!")
elif total >= 50:
    print("Shipping is $5.")
elif total >= 30:
    print("Shipping is $10.")
else:
    print("Shipping is $13")
```

In the example to the left, how many pathways are defined?

Yeah, 4 paths! Let's go through this code using a concrete example: pretend that our `total = 35`. First, we check the `if` statement. `35 < 75`, so we ignore the rest of the `if` statement. Next, we go down to check the first `elif` statement. Again, `35 < 50`, so we ignore the rest of this statement and go down to check the second `elif` statement. `35 >= 30` so we have a hit! We execute the rest of this `elif` statement and thus print out "Shipping is \$10" to the console.

Imagine if all of these were `if` statements instead of `elif` statements. What would happen if we had a total of, say, \$80? Our computer would check each condition sequentially and print out all of the first three statements because all of the expressions would evaluate to `True`. Having `elif` statements ensures that our computer only takes *one* path.

Exercise 4.13) In the example above, what would happen if our total equals a negative number (not a realistic situation, but in code there is a possibility that this could happen accidentally)? Debug the example to account for this error.

Exercise 4.14) Come up with your own `if/elif/else` statement using any scenario you want! It might help to refer back to Exercise 4.12. Your solution should include at least 1 `elif` block. To challenge yourself, include boolean operators when creating your conditions!

Exercise 4.15) What is the difference between an `if` statement and an `elif` statement? What about an `elif` statement and an `else` statement?

Lesson 5: Loops

Alright. Referring back to the previous example in Lesson 4, imagine you're calculating the correct shipping fee for a purchase transaction using conditional statements. You need to do this 3 times, for 3 different totals. Alright, cool, let's copy/paste the code block 3 times! Problem solved. But what about if you needed to do it 10 times? 50? 100+?

Personally, my fingers would probably fall asleep doing that; plus, I'd most likely miscount and accidentally only copy/paste about 99 times instead of 100. One way we can avoid this tedious repetition is by using something called *loops* in programming! Loops take a block of code and repeat it for us, without the need for manual copy/paste. That way, we can *reuse* code easily, thus making our code "cleaner" (nicer to read and simpler), and saving a lot of time and efficiency on our part as programmers.

For Loops

```
for i in range(10):
    print("Hello!")
# Prints "Hello!" 10 times
```

Supposed we wanted to print "Hello!" out to the console 10 times. The code block on the left demonstrates how we might do this using a *for loop* instead of repeating a `print("Hello!")` statement 10 times.

The syntax of a `for` loop is as follows to the right. The first line defines our `for` loop, with the code block that we want to repeat *indented* inside the `for` loop.

```
for [temporary_var] in range([# of times]):
    # do something
```

Here, `[# of times]` represents the **number of times we want to repeat our code block**. This should be a numeric, positive, whole number. Our `for` loop takes our code block and cycles through it `[# of times]`. We refer to each cycle as an *iteration*. Thus the `for` loop *iterates*, or runs, through our code block `[# of times]`. `temporary_var` is a temporary variable that we use to represent whatever iteration that we are currently on. Every time the `for` loop goes through an iteration, the temporary variable updates to reflect the current iteration. You can think of the temporary variable like a current state!

The temporary variable can be named anything, but best practice is to name it in the context of what we're doing (similar to variables). The temporary variable also does not have to be defined (AKA initialized or set to anything). So in our first example where we printed "Hello!" ten times, the "i" in the `for` loop could actually be any letter like "a", "j", etc. Generally, computer scientists use the letter "i" because of something called "lists" in programming, which is a more advanced concept that won't be covered in this packet. Lists are much like the kinds of numbered lists we have in real life (i.e. grocery lists, todo lists). However, programmers typically number lists starting at 0. So, for a list of 10 items, the items would be numbered from 0-9. We call these numbers "*indices*" (plural of *index*), hence the "i" letter.

In a `for` loop, we are essentially iterating over a numbered list starting from 0! Here's a diagram to help visualize this, referring back to our first example again:

i	0	1	2	3	4	5	6	7	8	9
iteration	1st	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
output	"Hello!"	"Hello!" "Hello!"	"Hello!" "Hello!" "Hello!"	"Hello!" ... (4 times)	... (5 times)	... (6 times)

We can think of a numbered list starting from 0, with each item being the code block that we want to repeat. We call each item on the list an “iteration”, much like how we call items on a grocery list “groceries”. In our example, with each iteration of the for loop, `i`, or the index, increases until it has iterated 10 times total. It’s like running through an actual list!

Exercise 5.1) What would the following code blocks output? (*Hint: for part b, pay attention to syntax & indentation!*)

a)

```
count = 0
for i in range(5):
    if(count % 2 == 0):
        print("Good morning")
    else:
        print("Good night")
    count += 1
```

b)

```
for count in range(25):
    print count
```

Exercise 5.2) Write a for loop that prints out your favorite food 3 times.

Exercise 5.3) Write a for loop that prints out the index if it is a multiple of 4 in the range 0-20.

Let’s say you were trying to find a particular number, like the number 4, in a range of numbers. Once you find it, you want to print out “Found it!”. To the right, we illustrate this scenario. However, do you notice something off about this loop?

```
for i in range(8):
    if(i == 4):
        print("Found it!")
```

It continues to iterate even after finding the number 4 and printing out “Found it!” While these empty iterations aren’t terrible, they do cost machine performance and efficiency. One way we can break out of a loop is by using the `break` keyword. When the program reaches a `break` statement, it will break outside of the `for` loop, thus avoiding empty iterations. One way this can be applied in a real-world situation is perhaps if we’re performing a matching algorithm with a huge quantity of data. When we find a match, we would want to break out of the loop instead of continuing to search through the massive amounts of data!

```
for i in range(8):
    if(i == 4):
        print("Found it!")
        break
    print("The end.")
```

Output:

Found it!
The end.

Conversely, there also exists the `continue` keyword, which allows us to “skip” iterations and go straight to the next iteration in the `for` loop.

```
for i in range(10):
    if(i % 2 == 0):
        continue
    print("Odd number: " + i)
```

Exercise 5.4) What is the code block on the left doing? What does it output to the console?

Exercise 5.5) Rewrite the code block so that it doesn’t use the `continue` keyword.

While Loops

While loops are another type of loops that repeat a code block *while* a certain condition is met (here's where our conditions are important!!). While loops, similar to *for* loops, iterate through a list. However, *while* loops run (i.e. repeat) until the condition is broken.

```
index = 0
while index < 20:
    print("I am out of ideas")
    index += 1
```

First, we initialize `index = 0`. While `index` is less than 20, we print out the message "I am out of ideas" and *increment* (increase) `index` by 1. As a result, this piece of code prints out "I am out of ideas" twenty times!

Notice how we didn't wrap our condition (`index < 20`) in parentheses. It's okay to not have parentheses here because we only have one condition we're checking. However, if you're combining multiple conditions to create a complex boolean expression, it's important to correctly wrap your conditions in parentheses to avoid confusion.

Take a second look at the example. Do you notice any similarities between the *while* loop and the *for* loop? In fact, we can translate this *while* loop into a *for* loop and vice versa! Why so? In a *for* loop, we are iterating through a range of numbers. In our example, we iterated a total of twenty times. If we were to translate this *while* loop into a *for* loop, that means our range would be 20!

Exercise 5.6) Translate the example *while* loop into an equivalent *for* loop.

Exercise 5.7) In our example *while* loop, we incremented `index` by 1 during each iteration. What would happen if we were to exclude this line of code from our loop?

Exercise 5.8) Translate Part A from Exercise 5.1 into a *while* loop.

Exercise 5.9) Sum up all the numbers from 1-50 using a *while* loop and store the result in a variable named `sum`.

We've seen that *while* loop and *for* loops can be interchangeable. So what's the point of having two different types of loops? While loops are most often used when we don't know how many iterations we need to fulfill a condition. For example, imagine that we want to find the number of times that a random number is divisible by 2 until the number is less than or equal to 1. If we knew this number, we could do the calculation. However, since we don't, we can use a *while* loop to do the calculation. Let's write how we would do this using *pseudo code*, or step-by-step instructions in plain language (we used pseudocode to create our algorithms in Lesson 2!):

```
get our number
initialize our count to 0
while our number is greater than 1
    divide our number by 2
    increment our count by 1
end
```

In our next lesson, we will learn how to get our random number. However, here we can see that our *while* loop will continue iterating until it accomplishes something, i.e. until our number ≤ 1 .

Nested Loops

Just like how we can repeat code using loops, we can actually iterate over *loops* as well! *Nested loops* are loops within loops. It may sound complicated, but let's break it down using an example:

outer loop value of i

<pre>for i in range(3): print("i = " + i) for j in range(3): print("j = " + j)</pre>	i	inner loop value of j		
	0	j = 0	j = 1	j = 2
	1	j = 0	j = 1	j = 2
	2	j = 0	j = 1	j = 2

When we execute this nested loop, we start at the outer loop, $i = 0$. The console prints out " $i = 0$ ". Then, we go into the inner loop, where $j = 0$. We go inside the inner loop and execute the inner loop's instructions: the console prints out " $j = 0$ " accordingly. Then, since we hit the end of the inner loop's instructions, we *increment the inner loop* and perform the instructions again, for $j = 1$. We repeat these steps again for $j = 2$. Then, since the next increment would be outside of the inner loop's range, we *jump out to the outer loop* and increment $i = 1$. The console prints out " $i = 1$ ". The inner loop has reset, so we start at $j = 0$ again and repeat what we did before for the inner loop. We can visualize this using the chart and going from left to right along the columns and down the rows. The outer loop starts with $i = 0$. Then the inner loop executes, incrementing j until we exit the inner range. We jump out to the outer loop (second row), increment $i = 1$, then execute the inner loop again starting at $j = 0$. We do this once more for $i = 3$.

Exercise 5.10) What would be the output of the above example?

You can think of this like each item in a list is another sub-list. So, when we iterate through the items on the list, we also iterate through each sub-list when we get to the item.

```
*
**
***
****
*****
```

Here's a more challenging example using nested loops: how can we reproduce the image on the left? Let's *decompose* this problem! First, take a look at the rows. We know that $i = 0$ on the first row, $i = 1$ on the second row, etc. There are 5 rows. We can write a `for` loop with `range(5)` that prints out one asterisk (*) with each iteration to get the first column of 5 stars.

for i in range(5): print("*")	Output
	*
	*
	*
	*
	*

So we know the outer loop iterates through rows. We need an inner loop to iterate through the columns now! First, let's make another table to help us visualize the problem. Remember that outer/inner loops iterate from left-right, top down.

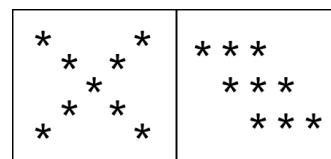
i = 0	j = 0	j = 1	j = 2	j = 3	j = 4
i = 1	j = 0	j = 1	j = 2	j = 3	j = 4
i = 2	j = 0	j = 1	j = 2	j = 3	j = 4
i = 3	j = 0	j = 1	j = 2	j = 3	j = 4
i = 4	j = 0	j = 1	j = 2	j = 3	j = 4

The cells in bold are the ones that represent our stars. Take a couple of minutes to look at the table. Do you notice a *pattern* between j and i ? Right, the last star of each row has $j = i$. All the stars below have that $j < i$. So that means we want to keep printing stars *as long as* $j \leq i$. Where do we print the stars? In the body of our outer loop or

the body of our inner loop? Remember that the inner loop *iterates along columns*. So, we would want to print the stars in the body of our inner loop! Take a couple of minutes to write the pseudocode of this *algorithm* (hide the below images before checking!!).

<pre> iterate through 5 rows iterate through 5 columns as long as j <= i print a star end </pre>	<pre> for i in range(5): for j in range(5): if(j <= i): print("*") else: continue </pre>
---	---

Exercise 5.11) Try to reproduce the two images on the right using asterisks, computational thinking, and nested `for` loops. *Hint: for the first diagonal crosses, think of sums!*



Lesson 6: Functions

Functions are blocks of code, or *sets of instructions*, that we can *call* to evoke. When we “call” a function, we run through the set of instructions once. So, functions allow us to repeat blocks of code without having to copy/paste code! What’s the difference between a function and loop, then?

Function: $2x + 1$	
Input (x)	Output
1	3
2	5
8	17

What’s special about a *function* is that we can actually feed “inputs”, or *arguments*, into the function and it’ll give us an output after performing its set of instructions on those inputs. It’s like a mathematical input/output machine (shown in the image to the left).

So, a function is a set of instructions that you can use over and over on different inputs. We actually already know one built-in Python function: `print()`. This function takes in a data type as an argument (i.e. “Hello”, 3, etc.) and outputs it out to the console.

```

def greeting_msg():
    print("Good morning!")
    print("How are you doing?")
greeting_msg() # Prints above lines

```

```

def function_name():
    # Some code
function_name() # Call function

```

Say that we want to print out the 2 lines “Good morning!” and “How are you doing?” every time a customer visits a website. Instead of repeating this code, we can put it in a *function* named `greeting_msg`. Every time we want to print these lines, we can call our function by using the syntax: `greeting_msg()`. The parentheses are super important!

Everytime we call `greeting_msg()`, our messages will print to the console. We define a function using the keyword `def`. Our function body, the code we want to repeat, should be indented. Otherwise, it doesn’t count as part of the function.

Functions can also take in inputs, or *parameters*. We use or operate on these parameters in our function bodies. On the right, `name` serves as a placeholder for whoever’s name we want to *pass into* `greeting_msg()`, similar to the input `x` in our initial

```

def greeting_msg(name):
    print("Good morning,", name)
    print("How're you,", name, "?")
greeting_msg("Beth")

```

math function table. We call name a *formal parameter*, and whatever string ("Beth") that we substitute into name the *argument*. The function assigns name = "Beth" at the beginning. If we called greeting_msg() with a different argument like greeting_msg("Anna"), then name = "Anna".

```
def greeting_msg(name, age,
major="CS", eye_color="brown"):
    print("Good morning,", name)
    print(name + " is " + age + "years
old and has " + eye_color + "
eyes.")
    print(name, "are you majoring in",
major, "?")
greeting_msg("Beth", 22, "Art",
"blue")
greeting_msg("Abby", 19)
```

Functions can actually take as many parameters as we want. So we can add: greeting_msg(name, age, major, eye_color) and more! However, however many parameters we have, we *must* provide arguments for all of them when we call the function: greeting_msg("Beth", 21, "Architecture", "Brown"). Otherwise, we will get an error. To avoid this error, we can use *default arguments*. We assign function parameters to default arguments, such as major="CS" and eye_color = "brown" in the example on the right. Now when we call greeting_msg(), we can choose to pass in these last two arguments, or leave them out like in

the example! If we leave them out, the function will use the default arguments. Default arguments *must be defined at the end* of a function signature (inside the parentheses). Otherwise, you will get an error.

Exercise 6.1) What is the output of the previous example?

Exercise 6.2) What is the output of the following code?

A)

```
def calc_avg(score_1, score_2 = 95,
score_3):
    print((score_1 + score_2 +
score_3) / 3)
print("Calculating average...")
calc_avg(75, 88)
calc_avg(79, 90, 89)
```

B)

```
def print_nums(num):
    for i in range(num+1):
        print(i)
print_nums(10)
print_nums()
```

Exercise 6.3) Debug/edit the code in 6.2A and 6.2B above so that the code runs.

Exercise 6.4) Write a function that takes in a parameter num and checks if num is a prime number. If so, it prints out "You found a prime." Else, it prints out "[num] is not a prime number." *Note: A prime number is a number that is only divisible by itself and 1.*

Functions can also *return* values to the user. We can store this value in a variable to be used later. To return a value, we use the `return` keyword. Note, we're returning *values*, not variables! As a result, if you return a value but don't do anything with it, then there is no point in calling the function in the first place. On the right, sum_nums sums all the numbers from 0-num, inclusive. sum_nums calculates and *returns* this sum, which we can store in variables like sum_10 and default_sum! What are the values of sum_10 and default_sum?

sum_10 = 55 and default_sum = 15! They store the returned sum from each individual call to sum_nums. Functions can also return *multiple values*. The function cubed(x, y) on the right takes in two numbers and

```
def sum_nums(num = 5):
    sum = 0
    for i in range(num+1):
        sum += i
    return sum
sum_10 = sum_nums(10)
default_sum = sum_nums()
```

```
def cubed(x, y):
    x_cubed = x ** 3
    y_cubed = y ** 3
    return x_cubed, y_cubed
x_3, y_3 = cubed(3, 8)
print(x_3)
print(y_3)
```

cubes them, returning both cubed values. We store the cubed values in the variables `x_3` and `y_3`. What does the code print out? (Answer: 27 and 512).

Exercise 6.5) Write a function that takes in a name (length ≥ 3 letters) and returns 3 values: the first letter, the middle letter, and the last letter. Note: Remember lists and indices? Indices number the items of a list. Like a list, the letters in a string are numbered starting from 0. We can find the length of a string using the function `len(string)`, i.e. `len("Anna")` would equal 4. We can refer to the first letter in a string as `string[0]`, the second letter as `string[1]`, the third as `string[2]`, the last as `string[len(string) - 1]`, and etc. Don't worry about the brackets syntax! Note 2: Remember that division yields a float value. Indices are integers. To change a float to an int, use `int([float value])`. This truncates everything in a float after the decimal point; i.e. $3.5 \rightarrow 3$, $4.75 \rightarrow 4$, etc.

Exercise 6.6) What would happen if in `cubed`, we accidentally forgot `y_3` and instead called `x_3 = cubed(3, 8)`?

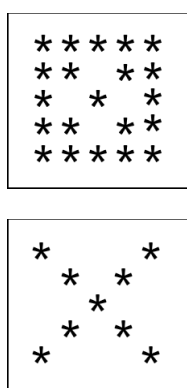
Exercise 6.7) Rewrite Exercise 6.4 using the `return` keyword. *Hint: when we use the return keyword, we actually don't have to return a value! When we use return without a value, it simply means to exit the function.*

Something we need to talk about regarding functions is *scope*. Going back to the `cubed` function, what if we called this function using the line: `x_cubed, y_cubed = cubed(4, 5)`? In fact, the code would actually still work! `x_cubed, y_cubed` inside `cubed` are *defined only inside the function scope*. Therefore, `x_cubed, y_cubed` that we define *outside* the function scope represent an entirely different set of variables. Similarly, if we tried to print out the parameter `x` or `y` of `cubed`, we would get an error because `x` and `y` are defined only in the function scope.

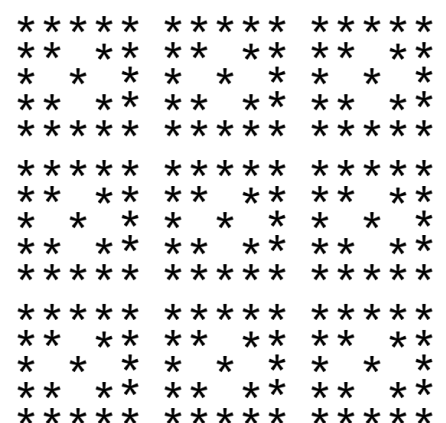
Exercise 6.8)

<p>A) What does the following code output?</p> <pre>def letter_grade(score): if(score >= 89): return("A") elif(score >= 80): return("B") elif(score >= 70): return("C") elif(score >= 63): return("D") else: return("F") print(letter_grade(77))</pre>	<p>B) Write a function named <code>count_letters</code> that takes in parameters <code>letter</code> and <code>word</code> and returns the number of occurrences of <code>letter</code> in <code>word</code>. Assume that <code>len(word) > 0</code>.</p> <p>a) <code>count_letters("a", "avocado")</code> returns 2</p> <p>b) <code>count_letters("e", "book")</code> returns 0</p> <p>c) <code>count_letters("m", "diamond")</code> returns 1</p>
--	--

Mini Project: Quilts (Mail us your creations!!)



Imagine that you want to create a quilt using code! You're not sure what dimensions your quilt will be, but you know it will be a square (i.e. 4×4 , 8×8 , etc). Using what you learned about computational thinking, variables, conditional statements, loops, and functions, create a program that can print out a quilt design in a dimension you choose. The individual patterns can all be the same or variations of patterns, some examples on the left. Your designs aren't limited to stars, feel free to use other characters as well!! The second image to the right is an example quilt!



Feedback

Hey, we've sadly reached the end of the packet. Thank you for sticking with me through all of it. I hope that you were able to gain something out of these lessons and that it wasn't *too* boring :) . I tried my best to include real-world examples where we would use these concepts, but I think some may have gotten lost in translation. However, I hope I have helped shed some light on how and where computer science can be used, and how powerful of a tool it can be!

We would love to hear what your thoughts were on this packet: What did you like? What didn't you like? What did you wish this packet included more of? What did it lack? How did your expectations of this packet differ from what it actually is? How can we improve this going forward? Additionally, please feel free to send any questions over. All Q's welcomed!

Exercise Answers

1.1) Some more examples: using AR/VR (augmented reality/virtual reality) for architecture to visualize what a built space would look like, creating assistance devices to call for help in the case of say an epilepsy seizure, creating art tech that can expand to 3D space.

1.2) Alan Turing is a notable computer scientist who created the Turing machine, the basis for the first computer. He also developed a test for artificial intelligence, still used today. During WWII, he helped decrypt Germany's Enigma machine, quickening the Allied victory. There's actually a movie about this accomplishment called "The Imitation Game". One inspiring computer scientist to me is Ada Lovelace, who was considered the first computer programmer!

1.3) Low-level programming languages are closer to machine code while high-level languages resemble more of human languages.

1.4) When you run a piece of code, it goes through the compilation process in which the compiler checks for any bugs, and if none are found, translates the program into machine language for the computer to execute.

1.5) IT focuses on maintaining & installing computer systems. Computer science focuses on developing these computer systems and making them run more efficiently. Information science focuses on solving problems created by computer systems and connecting people and CS.

1.6) For me, the answer would be: `print ("Hello Anna")`

2.1) Some tasks we can decompose this problem into: what kind of app is being created, who is the target audience, what the brand will look like, how the user will navigate the app,

what problem the app is trying to solve, how will you market your app, etc.

2.2) Shift = 12. Answer: THE CAESAR CIPHER WAS NAMED AFTER JULIUS CAESAR

2.3) Each number represents the sum of the two numbers in front of it! i.e. $2 = 1 + 1$, $5 = 2 + 3$. Formula: $a_n = a_{n-1} + a_{n-2}$ with a_n representing the current number, a_{n-1} representing the previous, etc.

2.4a) One way to decompose the PB&J problem is: A) Take out two pieces of bread. B) Take out the PB&J. C) Spread peanut butter on one slice of bread. D) Spread jam on one slice of bread. E) Put the two pieces of bread together.

2.4b) One pattern we can identify is steps C & D above – both involve spreading something over a slice of bread. 1 way we could make this into a formula: Spread x on one slice of bread, with x representing the type of spread we want.

2.5) Our solution may be incorrect or inefficient!

2.6) If we don't use abstraction, we could end up with an incorrect solution to our problem. For example, referring back to the essay structure example, if we didn't use abstraction we might think all body paragraphs are exactly 6 lines long with the same topic sentence. Using abstraction helps us form a clearer model of an essay!

2.7) Answers vary. Some examples: We assume all dogs have tails, fur, four legs, two ears, etc. (in real life this isn't true! But for the sake of simplicity we assume). Some specific details we don't need to focus on include whether the dog is muscular or not, whether it's tall or short, how long its tail is, how loud it barks, etc.

2.8) An example could be: 1) Rolling on your side. 2) Snoozing for 5 minutes. 3) Waking up. 4) Stretching for 5 min. 5) Drink

a glass of water. 6) Journaling for 5 minutes. We can use pattern recognition and abstraction to simplify this by taking a trend and making a formula out of it: in 3 steps, we do *something* for 5 minutes. We can simplify this to: do [x] for 5 min.

3.1) Answers vary. My answer would be: `print("If they don't give you a seat at the table, bring a folding chair.")` From Shirley Chisholm, the first Black woman elected to the US Congress.

3.2) A) Reason #2: helping other people reading the code understand what it does/means. B) Reason #1: providing context for a piece of code. The comment explains what is the purpose of the variable `num_states`. Without the comment, we would be left wondering – states for what? A game? The evaporation process?

3.3) `my_age = 18 // print(my_age)` (Using `//` to represent a new line)

3.4) Good morning Anna

3.5) `my_name = "Anna" // favorite_animal = "whale" // print("My name is " + my_name + " and my favorite animal is " + favorite_animal)`

3.6) `month = "December" // date = 7 // print("Today is", month, date) // # Note: using print this way automatically adds spaces!`

3.7) A) `SyntaxError` @ line 1: the quotes around the first string should be the same (single or double). B) `SyntaxError` @ line 2: missing a parentheses at the end. C) `NameError` @ line 4: Python can't recognize the string and thinks it should be defined in the program. To fix, put quotes around "I like rainy weather too".

3.8) `print(11 ** 2 % 2 == 0)`
If it's even, dividing by 2 should yield a remainder of 0

3.9) `print(83 % 5)` # This gives the remainder, or the people left in the last group! Answer: 3

3.10) Multiple answers. Example answer:

```
steak_story += "I tried again the next day, and it worked!" // random_num = 5 // random_num += 3 // print(steak_story, "It took me", random_num, "tries.") // # You could have also used concatenation to print it out!
```

4.1) If I'm in a good mood today, I'll sing in the shower. Else, I'm going back to sleep.

4.2) Varies. Examples: I know how to sew clothes (True), my name is Marie (False), I know 4 languages (True)

4.3) Opinions wouldn't qualify as boolean expressions. For example, "Ramen is the best thing in the world."

4.4) A) $(2 \neq 2) \rightarrow \text{False}$, B) $(4 \leq 10) \rightarrow \text{True}$, C) $(50.8 > 10) \rightarrow \text{True}$

4.5) Varies!

4.6) `test_bool = "true"` # It is type string

4.7) `test_bool = True`

4.8) Varies!

4.9) A) $(\text{True and False}) \rightarrow \text{False}$, B) $(\text{False or True}) \rightarrow \text{True}$, C) $((\text{not True}) \text{ and True}) \rightarrow \text{False}$, D) $((\text{not False}) \text{ or True}) \rightarrow \text{True}$, E) $((\text{not (False and True)}) \text{ or } ((\text{not True}) \text{ or False})) \rightarrow \text{True}$

4.10)

```
total = 45
if(total > 50):
    print("You get free shipping!")
else:
    print("Shipping is $5.")
```

4.11)

```
if(grade >= 70.0 or final_score >= 85.0):
    print("Pass")
else:
    print("Fail")
```

4.12) Varies. Example: If it's 11PM, then I'll go to sleep. Else, I'll draw for a little bit.

4.13) If our total was negative, our code would print out "Shipping is \$13". We can debug this by changing the first `if` statement to an `elif` and adding the `if` statement:
`if(total < 0): print("Ineligible.")` We want to put this check at the top so that if the total is ineligible, then the computer doesn't run through the conditionals unnecessarily.

4.14) Varies. Example answer:

```
if(time < 0 or time > 24):
    print("Ineligible time.")
# Using military time for simplicity
elif(time == 23):
    print("I'll go to sleep.")
elif(time > 10 and time < 23):
    print("I'm awake!")
else:
    print("I'm sleeping, go away!")
# Note: this is legal code since each conditional only has a one-line body
```

4.15) An `if` statement conditions on an independent path, while an `elif` statement goes hand-in-hand with an `if` statement and provides an alternative path. Similarly, an `else` statement goes hand-in-hand with `if`, but forces the computer to take the `else` path in the case that the `if` statement evaluates to False.

5.1) A) Good morning // Good night // Good morning // Good night // Good morning

Note: we increment `i` in this loop, but this is redundant code b/c `for` loops automatically increment. B) `NameError`: must put parentheses around the `count` variable. Once parentheses added, will output all the numbers from 0-24.

5.2) Varies. Example answer:

```
for count in range(3):
    print("Noodles!!! <3")
```

5.3) for `index` in `range(21)`:

```
    if(index % 4 == 0):
        print(index)
```

5.4) It loops through all the numbers from 0-9 and prints out all the odd numbers.

5.5) for `i` in `range(10)`:

```
    if(i % 2 != 0):
        print("Odd number: " + i)
```

5.6) for `index` in `range(20)`:

```
    print("I am out of ideas")
```

5.7) If we excluded the index increment, we would have something called an *infinite while loop*. An infinite while loop keeps running and never ends, while can slow down your computer and may break your program. It is vital to avoid infinite loops!! Usually, you can exit an infinite loop by refreshing the browser, or clicking a "stop" button in an IDE.

5.8)	5.9)
<pre>count = 0 while(count < 5): if(count % 2 == 0): print("Good morning") else: print("Good night") count += 1</pre>	<pre>sum = 0 count = 0 while(count <= 50): sum += count print(count)</pre>

5.10) `i = 0 // j = 0 // j = 1 // j = 2 // i = 1 // j = 0 // j = 1 // j = 2 // i = 2 // j = 0 // j = 1 // j = 2`

5.11)

For the diagonal crosses:	For the stairs:
<pre>for i in range(5): for j in range(5): if(i == j or i + j == 4): print("*")</pre>	<pre>for i in range(3): for j in range(5): if(j >= i): print("*")</pre>

6.1) Good morning, Beth // Beth is 22 years old and has blue eyes. // Beth, are you majoring in Art?
 Good morning, Abby // Abby is 19 years old and has brown eyes. // Abby, are you majoring in CS?

6.2) A) Error: all default arguments must be defined *at the end* of a function signature. B) Error: `print_nums()` doesn't pass in an argument.

6.3) A) Move `score_2` param after `score_3`. B) Either: add a default parameter to `sums`, or pass in an argument to our empty call.

6.4) We use `found_prime` to keep track of if `num` is a prime. We assume that `num` is a prime (`found_prime = True`), and throughout our `for` loop, if `num` is divisible by any number between 2-(`num-1`), then that means it isn't prime.

```
def is_prime(num):
    i = 2
    found_prime = True
    for i in range(num):
        if(num % i == 0):
            print(num, "is not a prime number.")
            found_prime = False
            break
    if(found_prime): print("You found a prime!")
```

6.5) `def name(name):`
 `return name[0],`
`name[int(len(name)/2)], name[len(name)-1]`
6.6) Our code would actually still run! `x_3 = (27, 512)`, which is something we call a *tuple* that won't be covered in this packet, but is legal code.

6.7) If `num` is divisible by any number between 2-(`num-1`), then it's not a prime. We print out the statement and exit the function. Otherwise, the `for` loop would execute all the way through, find no divisors, and print out the last statement.

```
def is_prime(num):
    i = 2
    for i in range(num):
        if(num % i == 0):
            print(num, "is not a prime number.")
            return
    print("You found a prime!")
```

6.8) A) Output: "C". B)

```
def count_letters(letter, word):
    count = 0
    # loop traverses all indices of word
    for i in range(len(word)):
        if(word[i] == letter): count += 1
    return count
```