THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A Unit Curriculum for Introductory Computer Programming

NICHOLAS STEVEN DORESKY
SPRING 2022

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Griselda Conejo-Lopez
Professor of Computer Science
Thesis Supervisor

Jesse Barlow
Professor of Computer Science
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

In these rapidly changing times, programming experience is becoming increasingly important for a growing number of jobs. Despite this, computer programming is not a required course at the secondary level, and therefore, many students graduate high school with little to no programming experience. In order to better prepare students for one of the fastest growing fields, I have researched effective methods of teaching computer programming. Drawing from this research and my own personal experiences as a student studying computer science, I have developed a unit curriculum for an introductory level computer programming class.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Professor Griselda Conejo-Lopez (Pennsylvania State University) for supervising this thesis.

Chapter 1: Introduction

Computer programming is one of the fastest growing fields. According to the Bureau of Labor Statistics, computer and information technology jobs are projected to grow 13 percent between 2020 and 2030. Even outside of strictly programming jobs, experience in coding is becoming increasingly important across the board, and many individuals will at some point in their professional careers need to know how to program. Despite these trends, computer science is not a required area of study for high school students. In the state of Pennsylvania, only 51% of high schools offer computer science classes (Code.org). Even schools that do offer rudimentary programming classes are likely to vary in quality and material since Pennsylvania has not developed a statewide plan for teaching computer science at the secondary level.

Speaking from personal experience, my high school did offer one computer programming class. It was my first exposure to coding, and I enjoyed it so much that it inspired me to pursue a computer science degree. However, in retrospect, I do acknowledge that my high school programming class had its flaws. During my freshman year of college, I was “introduced” to programming for a second time, through classes that were standardized and professionally developed.

As I continued my educational journey, I began to realize that I do not want to be a professional software developer, typing code for the rest of my life. However, I still have a strong passion for the area of computer science, and my interests began to shift towards teaching. My professional plans are to get my teaching certification in order to teach computer science and mathematics at the secondary level. This thesis is the union of my current studies and my future

aspirations: a unit curriculum for a secondary level computer programming course. Drawing from my experiences as a student of computer science, as well as research into pedagogical practices, I present the high school programming class that I would have wanted, and that I will teach in near the future.

Chapter 2: A Choice of Language

The first question I had to answer when developing this course was, “What programming language will it be taught in?” The debate surrounding the best first programming language is hotly contested and has no clear answer. I am in the unique position of having taken multiple introductory programming classes taught in different languages. My first high-school programming class was taught in C++, and my first at college was taught in Python. There are of course numerous academic papers written in favor of any given language. Both my personal experience and the research conducted have led me to pick Python as the language of choice for this course.

Above all else, I believe Python is the ideal introductory programming language due to its simplicity. Python was designed to be easily understandable yet powerful, requiring less code to achieve similar results to other languages. Python code is intentionally easy to read, especially for new programmers. Take the most ubiquitous program in any introductory computer science class for example, the “Hello, World” program. It is usually the first lines of code one ever writes when learning a new language, and its purpose is to familiarize the programmer with the syntax of the language. The “Hello, World” program in C++ contains around seven lines of code and includes a library, a namespace, and a function, three complex topics that are not covered until much later in the course, or quite possibly, not at all (Jenkins 4). The Python equivalent, as seen in the first lesson of this curriculum, requires only a single line of code.

Python is easier to understand because there is less syntactical overhead to worry about. Students learning C++ will undoubtedly wonder why `<iostream>` is being included or why `0` is being returned. Students who forget or neglect to include all of these tedious extra lines of code will run into enigmatic errors that inhibit the learning process. An abundance of syntax errors is

frustrating for the teacher and discouraging for the students, and thus a simple language allows new programmers to get up and running as fast as possible. In his argument for Python as a first language, John M. Zelle noted that, “The first classes should not be about language, but rather about computer science and, fundamentally, design. Trying to teach a complex language inherently detracts from that goal, since students must spend more time mastering the language and, hence, less time mastering other material.” The most important skill learned in an introductory computer science class is problem solving, not proficiency in any particular language. With how easy it is to work with, Python lends itself perfectly to that goal.

The main downside to Python is that it is not as robust as some other programming languages. It is undeniable that other common first languages such as C++ and Java are faster and more powerful than Python. However, these factors are irrelevant in the context of an introductory programming class. The features that make the aforementioned languages more robust than Python are advanced topics that are beyond the scope of such a class, and which students will never find themselves needing (Jenkins 2).

For the longest time, C++ and Java were the most popular introductory programming languages because they were the industry standard. If you’re going to learn a language, why not learn the most common, in-demand one? While this still holds true, Python has grown tremendously in commercial use in recent years. According to K. R. Srinath, “On Angel List, Python is the 2nd most demanded skill and also the skill with the highest average salary offered” (354). Python is in demand now more than ever, and we can only expect that demand to keep rising. The inherent advantage of learning C++ or Java instead of Python is becoming less pronounced, making the case for Python as a first language stronger.

I chose to teach this class in Python because of its simplicity, allowing students to spend less time learning laborious conventions of a language and more time solving unique problems as amateur computer programmers. It is also plenty powerful enough to do everything this class requires, and it is one of the fastest growing languages in the industry.

Chapter 3: Course Outline

This course is designed to be taught at the high school level with an 18-week semester schedule. The course material is broken up into "lessons." Each lesson does not correspond to a single day's lecture. The lessons can be thought of as general topics to be covered and most will span 1-2 weeks.

Week	Topic
1	Getting to know Python, print statements
2	Variables
3	Variables cont.
4	Operations on variables
5	Comments
6	Booleans
7	Booleans cont.
8	Lists
9	Lists cont.
10	If statements
11	While loops
12	For loops
13	For loops cont.
14	Functions
15	Functions cont.
16	Functions cont.
17	Flex week (catch up if behind, or show off something beyond the curriculum if time permits)
18	Review and final assessment

Table 1: Course Outline

Chapter 4: Lesson Plans

Lesson 0: Preparations

This course is designed to be taught in a high school with a traditional computer lab setting. Before the first day of classes, put in a request to the technology department to have Python installed on every computer in the lab. For methods of installation, see this tutorial: https://realpython.com/installing-python/. On a Windows computer, follow the instructions listed under, “How to Install From the Microsoft Store.” For a Mac computer, follow the instructions listed under, “How to Install From the Official Installer.”

These Python downloads come with the integrated development environment IDLE bundled in. For the purpose of this class, IDLE will be a sufficient workspace. Compared to other IDE’s it is very basic, but that can be a beneficial trait in an introductory programming class. The lack of fancy buttons and features makes IDLE less likely to confuse or distract amateur programmers. For the sake of simplicity, I do not recommend installing any other integrated development environments. With Python and IDLE installed on every computer, you are ready to begin coding as soon as classes begin.

Other Options

If the class is being taught in a traditional classroom rather than a computer lab, then it is likely that every student has a school-provided laptop. In that case, perform the same installation on the enrolled students’ laptops before the first day of classes. In the unlikely case that the school is not providing computers of any kind for this class, and students are expected to bring their own device, they will have to perform the installation themselves. Set aside the first day of class to explain and assist in the installation process. This is not the ideal option because it cuts

into a day of class time and issues are bound to occur. It is unlikely that an entire class of students successfully download and install any piece of software on the first try. It is preferable if this step can be left to the tech department.

Lesson 1: Getting to know Python

Goal

- Write and execute print statements in Python

Introduction

Introduce students to an integrated development environment (IDE) and show them its most basic features such as where to type code, save a project, and run a program.

Lecture

The first project a programmer tackles when learning a new language is the “Hello, World!” program. It is meant to be simple and helps the programmer get used to the conventions of that specific programming language. Python is one of the easiest programming languages to learn, and this is reflected in its “Hello, World!” program, requiring only one line of code.

Figure 1: The code for a "Hello, World!" program (left) and the output (right)

Exercise

Write a “Hello, World!” program on your computer, copying the code shown above. Then, modify your code in order to print “Goodbye, World!”.

Lesson 2: Variables

Goal

- Define and use variables in Python
- Understand the different types supported by Python

Motivation

The print statements we have learned about thus far are useful, but the number of things we can do with them is limiting. Computers are capable of so much more such as storing data, retrieving data, and solving math problems.

Lecture

Variables are a way to store and use data in your computer programs. A variable is created by giving it a name and assigning it a value using the equals sign, like in the example below.

```
myvar = 3
```

Figure 2: Creating a variable called "myvar" and assigning it the value 3

The data that a variable holds can be changed, hence the name “variable.” The lines in a program are executed in order from top to bottom, so the variable will hold the data that was most recently assigned to it.

```
myvar = 3

myvar = 7
```

Figure 3: Creating a variable "myvar" with initial value 3, then changing its value to 7

Data in Python comes in different forms which we call “types.” So far, the data we have seen are whole numbers (3 and 7). These are called integers, or **ints** for short, and are one of the types supported by Python. We can also work with decimal numbers, such as 4.3 and -16.1778,

which we call floating point numbers, or **floats** for short. It would be limiting if we could only work with numbers, so another type lets us store words, names, or just about any string of characters on the keyboard you can think of. Thus, the final type of variable is aptly named **string**.

You do not have to explicitly state the type of a variable when coding in Python. There are just certain conventions you must conform to:

- Floats should be written as decimals or fractions, using the . and / symbols respectively.
- Strings must have "quotation marks" around them. You can use 'single' or "double" quotes.

```
w = 4        # This is an int
x = 11/16    # This is a float
y = 3.2      # This is also a float
z = "dog"    # This is a string
```

Figure 4: Examples of the 3 types of variables

Last time we learned about print statements. These can be used with our variables to display the value they hold.

```
myInt = 15
print(myInt)
```
```
15
```

Figure 5: Printing a variable outputs the value that it holds

Exercise

What do you think this code will output? Keep in mind the location of print statements, the fact that variables can be changed, and the idea that code is executed in order from top to bottom.

```
x = 5
y = 12
x = 20
print(x)
x = "cat"
print(x)
print(y)
```

Lesson 3: Operations on Variables

Goal

- Expand the use of variables with the basic Python operators

Motivation

In the last lesson, we learned how to create variables, change the values they hold, and output those values using a print statement. What are some other things we may want to do with integers, floats, and strings? Open up discussion to the class. We may want to combine words to make sentences or subtract two numbers to find their difference.

Lecture

The plus operator (+) will add two numbers. It can be used on ints or floats.

```
x = 5
y = 12
print(x+y)
```
```
17
```

Figure 6: Adding two integers

You can also subtract, multiply, and divide numbers.

```
x = 12
y = 3
total = x + y
difference = x - y
product = x * y
quotient = x / y
print(total)
print(difference)
print(product)
print(quotient)
```
```
15
9
36
4.0
```

Figure 7: Four basic arithmetic operations

The plus operator has a different function when used on strings. It will attach the second string to the end of the first to create one big string. This is called string concatenation.

```
x = "water"
y = "melon"
print(x+y)
```

```
watermelon
```

Figure 8: String concatenation

Do you think the -, *, and / operators also work with strings? Discuss with your neighbor. Then try it out and discuss the results.

Exercise 1

Your friend is trying to write a program that, given a subject and predicate, combines the two to create a sentence. However, since concatenation simply attaches the second string to the end of the first, there is a space missing between the two. How would you modify his code, shown below, to generate grammatically correct sentences?

```
subject = "The students"
predicate = "worked hard on their assignment."
sentence = subject + predicate
print(sentence)
```

```
The studentsworked hard on their assignment.
```

Exercise 2

What do you think this code will output? This may look like some illegal mathematical operation, but keep in mind that the = symbol is an operator that assigns the value on the right to the variable on the left. Test it out for yourself and try to understand what is going on. This is a very common practice in computer programming.

```
x = 5
x = x + 1
print(x)
```

Lesson 4: Comments

Goal

- Understand how comments work, why they are useful, and when to use them

Motivation

As we improve as coders and start to write longer programs, we might reach the point where it becomes hard to look at the code and quickly comprehend what it is doing. This is especially troublesome when you are not the original author of the program. In the professional world, we do not want to spend hours or even days trying to explain what a program does every time a new developer joins a project. This is where comments become an invaluable resource for programmers who want their code to be easily readable for themselves as well as others.

Lecture

Comments are lines in your code that are not run when the program executes. They are solely there to be read by humans; the computer effectively ignores them. Put comments in your code to explain what is going on or ignore unwanted code that you do not want to completely get rid of. Comments are written by adding a preceding # symbol. Everything after the # will be commented out until the end of the line.

```
# This program determines whether
# two values are equivalent,
# printing True if they are equal
# and False otherwise.
x = 16
y = 34
print(x == y)
```

```
False
```

Figure 9: Add comments to explain what a block of code does

```
print("This line will be printed")
#print("This line will not be printed")
print("This line will also be printed")
```

```
This line will be printed
This line will also be printed
```

Figure 10: Comment out lines of code to prevent them from executing

Exercise

What do you think this code will output? Run it to find out.

```
x = 15
y = 9
z = 2

x=x+y
z=3
x=x/z
print(x)
```

Comment out one line to make the output 12. Comment out a different line to make the output 5. Notice how big of a difference just one line of code can make in a computer program.

Lesson 5: Booleans

Goal

- Introduce Boolean values
- Compare variables and store the result in a Boolean variable

Motivation

We have gradually been building on our ability to work with variables. First, we learned how to create, change, and print them. Then we learned how to perform operations on them. Suppose we want to answer logical questions about our variables, such as "is integer A greater than integer B?" or "does this string contain the letter P?". You may notice these are yes or no questions, which means the computer can easily store the answer as "true" or "false".

Lesson

Booleans are special variables that can hold one of exactly two values: True or False. Booleans usually arise as the result of some logical operation, but you can also directly create them, as shown below. Note that when referring to the Boolean values True and False, you must capitalize the first letter. Your IDE will usually color keywords, including Booleans, a different color, so it is easy to tell when you have written them correctly.

```
x=True      # x is a Boolean variable holding the value True
y=false     # This will cause an error. Boolean values are always capitalized.
            # Notice how the Boolean value is tinted orange.
```

Figure 11: Boolean values are always capitalized

Comparing the values of two numbers (int or float) will return a Boolean. Python supports the basic comparison operators you know from math class (greater than, less than, equal to, etc.).

```
x = 5
y = 3.2
print(x > y)        # Is x greater than y?
print(x == y)       # Is x equal to y?
```

```
True
False
```

Figure 12: Comparing two numbers returns a Boolean answer

Note the double equals sign when comparing two values for equality. As you know well by now, the single equals sign is used to assign a value to a variable. This is present in lines 1 and 2 of the code above. We do not want to assign x the value of y, so it needs a different symbol, hence the double equals sign. This is one of the most common errors, especially for new programmers.

Python Comparison Operators	
Syntax	**Meaning**
x == y	**x equals y**
x != y	**x does not equal y**
x < y	**x is less than y**
x <= y	**x is less than or equal to y**
x > y	**x is greater than y**
x >= y	**x is greater than or equal to y**

Table 2: Python Comparison Operators

Exercise

What do you think this code will output? Write down your answer, then write the code and run it to find out if you were right.

```
x=4
y=7
print(x>=y)
x=x+3
print(x>=y)
z=12
print(x+y>z)
```

Programming Assignment

- Programming Assignment 1 released. (see Chapter 5 for details on programming assignments)

Lesson 6: Lists

Goal

- Store different types of data in lists
- Perform operations on lists such as adding, removing, and sorting values

Motivation

We are becoming quite proficient at working with variables in Python. As we introduce more and more variables into a program, we may want a way to group together multiple variables in one place. For example, you may want to create a list of numbers. To this list you may want to do things such as sort the numbers from least to greatest, insert more numbers, or remove numbers you no longer want in the list. Python has built-in list functionality, and it is capable of all these things and more.

Lesson

A list is a data structure that allows multiple elements to be stored together and accessed through a single name. A list is created just like any other variable: you need to give it a name and then assign it a value with the equals sign. Items in the list are stored inside square brackets and separated by commas. The elements of the list can either come from variables (x and y in the example below) or be raw data values (40 in the example below).

```
x=3
y=12.4

list_of_numbers = [y, 40, x]

print(list_of_numbers)
```

```
[12.4, 40, 3]
```

Figure 13: Creating a list of numbers

Items in a list have a number that represents their position in the list. Just like if you are standing in a line, you can give a number to represent each person's position in line. This person

is first, that person is second, I am third, etc. In computer programming, this number is called an **index.** You can access a particular element of a list by writing the list's name followed by square brackets containing that element's index. Note that indices start counting at **0, not at 1!** So the first element of a list is retrieved with `list_name[0]`. Likewise, `list_name[1]` will return the second element of the list and so on. This is another common point of confusion for new programmers, but once again, it is standard in almost all programming languages and is something you must learn to accept.

```
x=3
y=12.4

list_of_numbers = [y, 40, x]

print(list_of_numbers)
print(list_of_numbers[0])
print(list_of_numbers[2])
```

```
[12.4, 40, 3]
12.4
3
```

Figure 14: Accessing list elements by indices

While you can retrieve an element from a list by its index, you can also modify an element of a list in a similar way. Once again, write the name of the list followed by square brackets containing the element's index. Then assign it a new value using the equals sign operator just like when we change the value of any other variable.

```
colors = ["orange", "red", "blue"]

colors[1] = "green"
print(colors)
```

```
['orange', 'green', 'blue']
```

Figure 15: Changing an element of a list

Being able to change an element of a list is nice, but it is also limiting. What if you want to add a new item to a list without replacing an existing one? Luckily, Python has methods that allow you to do this as well. Adding an item to the end of a list is called "appending." Adding an

item somewhere in the middle of the list is called “inserting.” Each of these actions has their own methods. They share a similar syntax, but the insert method requires you to include the index as well.

```
colors = ["orange", "red", "blue"]

colors.append("purple")
colors.insert(1, "green")
print(colors)
```

```
['orange', 'green', 'red', 'blue', 'purple']
```

Figure 16: Add items to a list with 'insert' and 'append'

As you may expect, if you can add items to a list, you can also remove items from a list. Use the “remove” method to remove an item with the given value. If you do not know the value of the item you wish to remove but you know the index it is located at, you can use the “pop” method to remove it. Use the “clear” method to remove all elements from the list.

```
colors = ["orange", "green", "red", "blue", "purple"]

colors.remove("red")
print(colors)
colors.pop(0)
print(colors)
colors.clear()
print(colors)
```

```
['orange', 'green', 'blue', 'purple']
['green', 'blue', 'purple']
[]
```

Figure 17: Three methods to remove items from a list

You can easily determine the number of elements a list contains, called the “length” of a list, by using the `len()` method. You can sort a list, alphabetically or numerically, using the `sort()` method. Finally, you can reverse the order of elements in a list using the `reverse()` method.

```
integers = [5, 3, 11, 92, 45, 8]

print(len(integers))
integers.sort()
print(integers)
integers.reverse()
print(integers)
```

```
6
[3, 5, 8, 11, 45, 92]
[92, 45, 11, 8, 5, 3]
```

Figure 18: Miscellaneous list methods

Python List Methods	
Syntax	**Meaning**
append()	Add element to end of list
clear()	Remove all elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the given value
extend()	Adds the elements of one list to the end of another list
index()	Returns the index of the first element with the given value
insert()	Inserts an element at the given index
pop()	Removes the element at the given index
remove()	Removes the item with the given value
reverse()	Reverses the order of the list
sort()	Sorts the list

Table 3: Python List Methods (adapted from w3schools.com)

Exercise

What do you think this code will output? It may help to go line by line and write what the list looks like after each individual step.

```
fruits = ["apple", "pear", "banana"]

fruits.append("watermelon")
fruits.pop(1)
fruits.insert(0, "tangerine")
fruits.sort()
fruits.reverse()
print(fruits)
```

Lesson 7: If Statements

Goal

- Expand our use of Booleans through if statements

Motivation

With Booleans, we are able to perform comparisons and get an answer in the form of True or False. But what can we do with this information other than print it out? It will often be the case that you want to run certain lines of code only if a particular condition is satisfied. For example, if you are coding a vending machine, you want to dispense a drink ONLY IF a certain amount of cash has been inserted. If statements will be an incredibly useful tool that will add depth and variety to your programs.

Lesson

If statements have several parts to them. They begin with the keyword “if”. This is then followed by a Boolean condition (expression that is either True or False) followed by a colon. Finally, there is the block of code that you want to be executed only if the condition was true. To differentiate these lines of code from other lines of code (that you want to execute no matter what), the dependent block is indented. See a simple example below.

```
print("This is the beginning of the program.")
condition = True
if condition:
    print("This is inside the if statement.")
print("This is the end of the program.")
```

```
This is the beginning of the program.
This is inside the if statement.
This is the end of the program.
```

Figure 19: An example if statement

If statements are the most complex pieces of code we have covered yet, so take your time trying to understand the above snippet of code and its output. What happens if you change the variable “condition” to False?

There are two other keywords that are often paired with if statements: “elif” and “else”. These cannot exist on their own and must be placed after an if statement. You can think of “elif” as a follow-up if statement. In the case where the first condition is not met, check this other condition. You can chain several elif’s together to check as many conditions as you like. You can use the “else” keyword as a sort of final option. In the case where none of the preceding if and elif’s were satisfied, then the block of code associated with else will be printed. Pairing if and else together is useful when we want to perform exactly one of two possible actions (Parlante 1).

```
x=12
y=5

if x < y:
    print(x,"is less than",y)
elif x == y:
    print(x,"equals",y)
else:
    print(x,"is greater than",y)
```

```
12 is greater than 5
```

Figure 20: If/elif/else example

Exercise

What do you think this code will output? Write down your answer, then write the code and run it to find out if you were right.

```
myList = [0, 5, 2, 8, 1, 3]
x = 3
if x > myList[x]:
    print("Hello")
else:
    print("Goodbye")
```

Programming Assignment

- Programming Assignment 2 released

Lesson 8: While Loops

Goal

- Explore a new use for the blocks of code we introduced along with if statements

Motivation

If statements allowed us to execute blocks of code under the condition that some Boolean expression is True. What if there is a block of code that we want to execute several times? We could continuously jump back to the beginning of the block and keep executing ONLY WHILE the condition is still True. Luckily, there is a special command for this action as well, and it is called a while loop.

Lesson

A while loop will continuously execute a block of code as long as a condition is True (W3Schools). It has a similar syntax to the if statements we are already familiar with. They begin with the keyword “while”, followed by a Boolean condition and a colon, and then have the indented block of code to potentially be executed. The only difference between the two is after the block of code is done executing in a while loop, the computer jumps back (or “loops”) to the beginning of the block and attempts to execute it again, given the Boolean condition is still True.

```
print("This is the beginning of the program.")
condition = True
while condition:
    print("This is inside the while loop.")
print("This is the end of the program.")
```

```
This is the beginning of the program.
This is inside the while loop.
This is inside the while loop.
This is inside the while loop.
This is inside the while loop.
This is inside the while loop.
This is inside the while loop.
This is inside the while loop.
This is inside the while loop.
This is inside the while loop.
This is inside the while loop.
```

Figure 21: An infinite while loop

In the above example, you may notice that the condition will always be True, meaning the program will never escape from the while loop. This is called an infinite loop and is something you typically want to avoid anytime you write a while loop. Usually, you will edit a variable that the condition depends on inside the while loop, such as incrementing a number.

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
print("This is the end of the program.")
```

```
1
2
3
4
5
6
7
8
9
10
This is the end of the program.
```

Figure 22: A while loop that counts from 1 to 10

Exercise

Write a program that counts down from 10 to 1, printing each number along the way. Implement this efficiently by using a while loop. Feel free to use the sample code from Figure 22 as a starting point. You will have to tweak some parts of the code, but the general form may end up looking very similar.

Lesson 9: For Loops

Goal

- Tie together our study of loops and lists with the concept of for loops
- Discuss the range method and understand its relation to for loops

Motivation

In Lesson 6, we introduced lists and various methods for performing operations on lists. So far we have no easy way of stepping through the list, element by element, and performing some action to each item along the way. Right away, this sounds like something that can be done with a loop: it is a series of actions that we want to perform a given number of times. In this case, the number of iterations is the number of items in the list. Thankfully, this process is made very easy with a new type of loop: the for loop.

Lesson

A for loop can iterate over a list and perform some action for each item it comes across. The syntax is a bit more complicated than a while loop, so have a look at an example and then read the explanation.

```
colors = ["red", "blue", "green"]

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

```
red
blue
green
```

Figure 23: Printing the elements of a list using a for loop

Similar to if statements and while loops, you start with the respective keyword, in this case “for.” This is followed by a variable that will take on the value of an item in the list on every iteration. This variable is called “i” in the above example, but it can be given any name.

The keyword “in” tells the computer that whatever comes next is what we will be iterating over, which is why it is followed by the name of the list.

Lists aren’t the only object you can iterate over using a for loop. You can also iterate over a range of numbers. This allows you to execute a block of code a particular number of times. The range method effectively creates a list of numbers from 0 up to but not including the specified number. This means `range(4)` equals (0, 1, 2, 3). This may seem unintuitive, but it is once again a consequence of programmers and computers starting their counting at 0 instead of 1. It works out though because (0, 1, 2, 3) contains 4 elements, so iterating over that range will still give us 4 iterations, which is what we ultimately wanted.

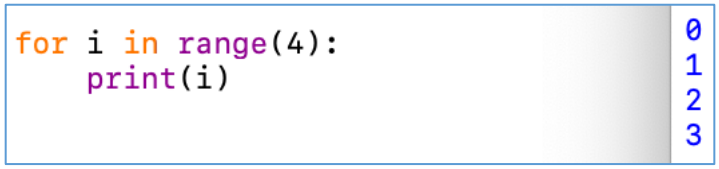

Figure 24: Iterating over a range of numbers

You can even iterate over a string, character by character.

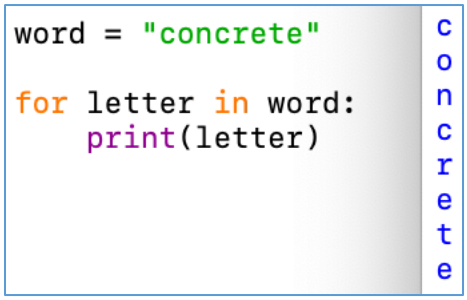

Figure 25: Iterating over a string

It is time to address another common mistake for new programmers. Suppose you wanted to iterate over a list and perform some actions to every element, such as doubling every item in a list of numbers. It would make sense to write code looking something like the following.

```
numbers = [7, 13, 22]

for x in numbers:
    x = x*2

print(numbers)
```

```
[7, 13, 22]
```

However, this does not output the expected results. What happened? The loop variable, x in this example, is only assigned a **copy** of the current element each iteration. Any modifications are done to the variable x, but not the element of the list itself. The original list is left unchanged. However, iterating over a list and doing a common operation to each element is something we may want to do. How can we achieve this? One easy solution is to iterate over the **indices** of the list and directly assign each element a new value.

```
numbers = [7, 13, 22]
list_size = len(numbers) # Should equal 3

# Range of 3 equals (0, 1, 2)
for i in range(list_size):
    numbers[i] = 2*numbers[i]

print(numbers)
```

```
[14, 26, 44]
```

Figure 26: For loop that doubles every element of a numerical list

Exercise

What do you think this code will output? Write down your answer, then write the code and run it to find out if you were right.

```
numbers = [17, 36, 3, 10, 8]
for number in numbers:
    print(number < 10)
```

Lesson 10: Functions Part 1

Goal

- Introduce functions
- Understand how functions differ from other blocks of code like if statements, while loops, and for loops

Motivation

If statements and while loops are very useful as they allow blocks of code to be executed under certain conditions. However, they are also limited in that they exist in one place in a program's code, and they can only run if the program reaches their line of code. Imagine you write a program with an if statement at the very beginning. Upon executing the program, once it reaches the if statement, it checks the condition, it may or may not execute the associated block of code, and then it moves on. After the program is finished with this if statement, it can never be reached again, at least not with our current tools. Functions are extremely convenient for a programmer as they are blocks of code that can be called upon to run from any point in a program.

Lesson

Similar to variables, functions must be defined before they can be used. To define a function, begin with the keyword "def", then give your function a name followed by opening and closing parentheses, and finally a colon. Then, similar to an if statement or while loop, indent the block of code you want the function to execute.

```
def example_function():
    print("Hello, World! This is a function!")
```

Figure 27: Declaring a function

Note that the above lines of code will never be executed on their own. A function can only run if it is called. A function is called by writing its name followed by the open and closed parentheses.

```
def example_function():
    print("Hello, World! This is a function!")

example_function()
```

```
Hello, World! This is a function!
```

Figure 28: Calling a function

It may not be clear in the above example, but it is important to note that the print code is not executed until after the program has reached the final line of code which proceeds to call the function. In a sense, the program flows backwards when the function is called, moving to an earlier line of code. This is behavior we have previously only seen in while loops. After the function finishes executing, the program will jump back to the line which called the function and continue to execute normally from that point. The following example will demonstrate these ideas. Take your time to trace the order of program execution, noting where control flow jumps to when the function is called and when it finishes.

```
print("This is the beginning of the program.")

# Define the function
def example_function():
    print("This is inside a function!")

print("The function has been defined, but not yet called.")
#Call the function
example_function()

print("This is the end of the program.")
```

```
This is the beginning of the program.
The function has been defined, but not yet called.
This is inside a function!
This is the end of the program.
```

Figure 29: Example of control flow with a function

Lesson 11: Functions Part 2

Goal

- Understand how to pass in and get out data from a function through the use of arguments and return values

Motivation

So far, functions may not seem very powerful. Consider the functions you are familiar with from math class, such as f(x) = 2x. This function is capable of taking an input, x, and returning an output, f(x). As it turns out, Python functions are capable of doing the same thing. In computer programming, the input to a function is called the "argument" and the output is called the "return value."

Lesson

To pass an argument into a function, place it inside the parentheses immediately following the function's name. Remember that the name of the function appears in the one place that you define it and in every place that you call it. The argument must be added in each of these places. If the number of arguments at the time of declaration does not match the number of arguments passed in when called, it will result in an error.

```
my_int = 99

def double(x):
    print(2*x)

double(4)
double(12.6)
double(my_int)
```

```
8
25.2
198
```

Figure 30: A function that doubles the numerical input

A function can take multiple arguments. They are still placed inside the parentheses after the function's name, and they are separated by a comma. The order in which they are placed **does** matter. The first value listed gets assigned to the first variable; the second value gets assigned to the second variable and so on. Why does this matter? Think of a subtraction problem. If you are told to subtract two numbers, a and b, the order makes a difference. Unless they are the same number, a-b will produce a different value than b-a. When you are defining and calling functions, make sure each value is getting assigned to the right variable.

```
number1 = 12
number2 = 5

def subtract(a,b):
    print(a-b)

subtract(number1, number2)
subtract(number2, number1)
```

```
7
-7
```

Figure 31: In multi-argument functions, the order of arguments matters

We now know how to provide a function with one or more inputs. Now let's see how the function itself can return an output. If you write the keyword "return" inside a function, then the value that follows will be sent back to the caller of the function. From there, this return value can be stored as a variable, computed upon, or printed.

```
number1 = 4
number2 = 11

def add(a,b):
    return a+b

total = add(number1, number2)
print(total)
```

```
15
```

Figure 32: A function with a return value

One function can be called from within another function. A function can even call itself! A function that calls itself is called a recursive function. Similar to the infinite loops that we have to be wary of when writing while loops, we also need to watch out for infinite recursion. This is when a function keeps calling itself indefinitely.

```
def double(num):
    num = num * 2
    return num

def half(num):
    num = num / 2
    return num


def fun1(x):
    if x > 10:
        x = half(x)
    else:
        x = double(x)
    print(x)

nums = [7, 64, 10]
for x in nums:
    fun1(x)
```

```
14
32.0
20
```

Figure 33: A function can call other functions

```
def countdown(x):
    print(x)
    if x>0:
        countdown(x-1)

countdown(10)
```

```
10
9
8
7
6
5
4
3
2
1
0
```

Figure 34: A function can call itself

Exercise

Implement the “increment” function. This function takes one argument in the form of a number and returns that number plus 1. For example, `increment(3)` should return 4.

Programming Assignment

- Programming Assignment 3 released.

Chapter 5: Programming Assignments

Programming Assignment 1: Operations on Variables + Booleans

Prompt

Suppose a Python savvy teacher is trying to automate some of the grading in his class. He gives three exams throughout the semester and a final at the end. However, if a student's average exam grade on the first three is at least a 90%, then they do not have to take the final. He wants an easy way to find out who can skip the final and who needs to take it. Write a program that, given three exam scores as input, prints True if the average score is at least 90% and False if it is less. Test your code with different exam scores to make sure it prints the desired result for averages both above and below 90. Also make sure your program is correct in the edge case where the average exam score is exactly 90%.

Sample Inputs and Outputs

Input (test scores)	Output
91, 88, 97	True
92, 88, 85	False
90, 90, 90	True

Programming Assignment 2: If Statements

Prompt

Let's continue to improve the automated grading program (Programming Assignment 1). Currently, the program prints True if the average score is at least 90% and False if it is less. Let's make the outputs a bit more intuitive for the user. Using an if/else statement, modify the program to print "The average is at least 90. Student does not need to take final." or "The average is less than 90. Student must take final." Additionally, the teacher wants to guarantee that any student who got a 75% or lower on any given exam must take the final. This takes precedence over the average score criteria, meaning in the case where a student performed excellently on two exams, but got below a 75% on the third, they **still must take the final.** Hint: Use the `min()` method to find the minimum of the three scores.

Sample Inputs and Outputs

Input (test scores)	Output
91, 88, 97	"The average is at least 90. Student does not need to take final."
92, 88, 85	"The average is less than 90. Student must take final."
90, 90, 90	"The average is at least 90. Student does not need to take final."
100, 73, 99	"The minimum is 75 or less. Student must take final."

Programming Assignment 3: Functions

Prompt

Write a function that calculates whether a positive number is prime or composite. A prime is a number which is only divisible by one and itself. Hint: Use the modulo operator (%) to get the remainder after the division of two numbers.

For example,

`7 / 3 = 2 Remainder 1.`

Therefore,

`7 % 3 = 1.`

If `a` is divisible by `b,` then `a % b = 0.`

For example,

`36 % 9 = 0`

Sample Inputs and Outputs

Input	Output
10	“10 is not prime”
7	“7 is prime”
49	“49 is not prime”
163	“163 is prime”

Chapter 6: Final Assessment

Introduction to Computer Programming – Final Exam

Part I: Multiple Choice

1. **Int**, **string**, and **float** are all Python _____.

    a. Comments

    b. Types

    c. Functions

    d. Loops

2. Which of these operations will result in an error?

    a. 12 + 5

    b. “water” + “melon”

    c. 8 + 6.5

    d. “dog” + 3

3. **True** and **False** are examples of _____ values.

    a. Boolean

    b. Int

    c. Float

    d. Range

4. Functions can take _____ argument(s).

a. 0

b. 1

c. More than 1

d. All of the above

5. Comments in Python start with the _____ symbol.

a. %

b. //

c. ~

d. #

6. A(n) _____ is able to iterate over the elements of a list, range, and string.

a. If statement

b. While loop

c. For loop

d. Function

Part II: Short Answer

7. Explain the difference between the single equals sign = and double equals sign == operators.

8. Explain the difference between if statements, for loops, and while loops.

9. Explain the difference between the `append()` and `insert()` list methods.

Part III: Code Comprehension

10. What will the following code output when run?

```
colors = ["red", "blue", "green"]
colors.append("pink")
colors.pop(0)

for x in colors:
    print(x)

for i in range(len(colors)):
    colors[i] = "gray"

print(colors[1])
```

11. What will the following code output when run?

```
def fib(a, b):
    return a + b

sequence = [0, 1]
n = len(sequence)

while n < 10:
    new_item = fib(sequence[n-2], sequence[n-1])
    sequence.append(new_item)
    n=n+1

print(sequence)
```

Chapter 7: Conclusion

In a field as rapidly developing as computer science, it seems natural for each high school to offer a modern, foundational introduction to programming class. However, since computer science education is not a requirement for graduation, many high schools do not offer any programming classes (Code.org). Even in schools that are offering introductory level classes, they may be outdated or taught in a less than ideal language for beginners. My research and personal experience have led me to the conclusion that Python is the best first language for new programmers. It is by far one of the simplest languages syntactically, making it easy and fun to learn. However, it is also becoming increasingly common in the commercial programming sphere (Srinath, 354), meaning students who decide to pursue further computer science education will have a meaningful foundation to build upon.

The unit curriculum is designed to introduce students to the fundamentals of computer programming through the welcoming lens of the Python language. Upon completion of this course, students will be adequate Python coders, but more importantly, they will possess a strong problem-solving philosophy and the ability to think like a computer programmer. The skills and mindset developed in this class can be applied to any future computer science classes that students may go on to pursue.

It is my recommendation that a course like the one outlined in this thesis be offered in every high school. After I graduate and go on to get my teaching certification, this will be a valuable resource for when I get to teach my first introductory computer programming course. I encourage school districts, especially those not currently offering programming classes, to implement this course. With proper computer science education, we can prepare students for the 21st century, and the age of technology it has brought upon.

BIBLIOGRAPHY

"Computer and Information Technology Occupations." *U.S. Bureau of Labor Statistics,* www.bls.gov/ooh/computer-and-information-technology/home.htm

"The Framework for Computer Science 7-12 Program Guidelines." *Pennsylvania Department of Education,* www.education.pa.gov

Jenkins, Tony. "The First Language - A Case for Python?" *Innovation in Teaching and Learning in Information and Computer Sciences,* vol. 3, no. 2, 2004, pp. 1-9.

Parlante, Nick. "If Statements and Booleans." CS106A, Stanford University, Fall 2004-05. Course handout. https://web.stanford.edu/class/archive/cs/cs108/cs108.1082/106a-java-handouts/HO32IfBoolean.pdf

Srinath, K. R. "Python – The Fastest Growing Programming Language." *International Research Journal of Engineering and Technology,* vol. 4, no. 12, 2017, pp. 354-357.

"Statewide High School Graduation Requirement." *Pennsylvania Department of Education,* www.education.pa.gov

"Support K-12 Computer Science Education in Pennsylvania." *Code.org,* code.org/advocacy/state-facts/PA.pdf

Zelle, John M. "Python as a First Language." https://mcsp.wartburg.edu/zelle/python/python-first.html

Zinth, Jennifer Dounay. "Computer science in high school graduation requirements." *Education Commission of the States,* files.eric.ed.gov/fulltext/ED556465.pdf

Academic Vita of Nicholas Doresky
ndoresky@gmail.com

Education

Pennsylvania State University, Schreyer Honors College
Computer Science, B.S.

Research

Thesis: A Unit Curriculum for Introductory Computer Programming
Thesis Supervisor: Griselda Conejo-Lopez

Experience

Summer 2019
Developer, ***Sperry Labs***
Webpage and database design

Fall 2020
Capstone Project, ***John Deere***
Collaborated with a team to develop an algorithm using artificial intelligence and computer vision to detect turf damage

Summer 2021
Technology Intern, ***Milton Area School District***
Checked student devices for damage and reimaged laptops

Awards and Honors

Recipient: Dreifuss & Hackenberg Family Fund Scholarships
Member: Tau Beta Pi, Engineering Honor Society