

## CSC 104 Lecture Notes

Prof. Christopher R. Merlo  
Nassau Community College  
cmerlo@ncc.edu

<http://www.matcmp.ncc.edu/~cmerlo/>

October 14, 2019





# Copyright

These lecture notes are © 2019 Christopher R. Merlo.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.





# Contents

<b>Copyright</b>	<b>iii</b>
<b>List of Figures</b>	<b>xi</b>
<b>List of Code Examples</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xv</b>
<b>About These Notes</b>	<b>xvii</b>
<b>Acknowledgments</b>	<b>xix</b>
<b>I Problem Solving</b>	<b>1</b>
<b>1 Course Introduction</b>	<b>3</b>
1.1 Course Strategies . . . . .	3
1.2 Daily Preview . . . . .	4
1.3 Interview Questions . . . . .	5
1.4 Logo Exercise . . . . .	5
1.5 Homework . . . . .	5
<b>2 Talking Like a Programmer</b>	<b>9</b>
2.1 Interview Question . . . . .	9
2.2 Vocabulary . . . . .	10
2.2.1 What Is a Computer Program? . . . . .	10
2.2.2 What Is an Algorithm? . . . . .	10
2.2.3 What's In a Computer? . . . . .	11
2.2.4 Kinds of Programming Errors . . . . .	12
2.2.5 How the Computer Reads Our Instructions . . . . .	12
2.3 Three Fundamental Components of a Program . . . . .	13
2.3.1 Sequential Statements . . . . .	13
2.3.2 Selection Statements . . . . .	13
2.3.3 Iteration Statements . . . . .	13
2.4 Programming Ethics . . . . .	14

2.4.1	General Principles . . . . .	14
2.5	Writing and Following Directions Exercise . . . . .	14
2.6	Homework for Day 3 . . . . .	14
<b>3</b>	<b>Matrix Logic</b>	<b>17</b>
3.1	What Is Matrix Logic? . . . . .	17
3.2	Example Matrix Problem . . . . .	17
3.3	Matrix Logic Exercises and Homework . . . . .	21
<b>4</b>	<b>Systematic Lists</b>	<b>23</b>
4.1	Interview Question . . . . .	23
4.2	Systematic Lists . . . . .	23
4.2.1	Why Use Systematic Lists? . . . . .	24
4.2.2	How Do We Create a Systematic List? . . . . .	24
4.2.3	Creating a Systematic List . . . . .	24
4.3	Characteristics of a Systematic List . . . . .	27
4.4	Creating Systematic Lists . . . . .	27
<b>5</b>	<b>Binary Numbers</b>	<b>29</b>
5.1	Review Exercise . . . . .	29
5.2	What Is a Number System? . . . . .	29
5.3	Binary Numbers . . . . .	30
<b>6</b>	<b>Introduction to Conditionals</b>	<b>31</b>
6.1	Decisions . . . . .	31
6.2	Conditions and Boolean Values . . . . .	31
<b>7</b>	<b>Nested Conditionals and Complex Condionals</b>	<b>33</b>
7.1	Nested Conditionals . . . . .	33
7.2	Complex Conditionals . . . . .	36
7.2.1	Boolean Operators . . . . .	36
7.3	Writing Conditional Statements . . . . .	37
7.3.1	The Drinking Water Example . . . . .	37
7.4	A Note About Grading . . . . .	38
<b>8</b>	<b>Debugging Conditionals</b>	<b>39</b>
8.1	Debugging Conditionals . . . . .	39
<b>9</b>	<b>Exam 1 Review</b>	<b>41</b>
<b>10</b>	<b>Exam 1</b>	<b>41</b>
<b>II</b>	<b>Programming in Python</b>	<b>43</b>
<b>11</b>	<b>Introduction to Python</b>	<b>45</b>

11.1	An Example Program . . . . .	46
11.2	How to Write and Run the Hello World Program . . . . .	46
11.3	Starting to Program in Python . . . . .	47
<b>12</b>	<b>Python Errors and zyLabs</b>	<b>49</b>
12.1	Python Errors . . . . .	49
12.1.1	Syntax Errors . . . . .	50
12.1.2	Runtime Errors . . . . .	50
12.1.3	Semantic Errors . . . . .	51
12.2	zyLabs . . . . .	51
<b>13</b>	<b>Variables and Expressions</b>	<b>53</b>
13.1	Variables . . . . .	53
13.1.1	Assignment . . . . .	53
13.1.2	A Variable Has One Value . . . . .	54
13.1.3	Identifiers . . . . .	54
13.2	Objects . . . . .	54
13.3	Two Kinds of Numbers . . . . .	54
13.4	Expressions . . . . .	55
<b>14</b>	<b>Modules</b>	<b>57</b>
14.1	Where Your Files Go When You Save Them . . . . .	57
14.1.1	The Department's Computers . . . . .	57
14.1.2	Your Computer . . . . .	58
14.2	Three Kinds of Python Programs . . . . .	58
<b>15</b>	<b>Strings and Numeric Types, and Writing a First Python Script</b>	<b>59</b>
15.1	Strings . . . . .	59
15.2	Numeric Types . . . . .	60
15.3	Writing a Python Script . . . . .	60
15.4	The Header . . . . .	60
15.4.1	The Shebang . . . . .	60
15.4.2	The Docstring . . . . .	61
15.4.3	The Rest of the Header . . . . .	62
<b>16</b>	<b>Tracing a Python Program</b>	<b>63</b>
16.1	Tracing a Python Program . . . . .	63
<b>17</b>	<b>The graphics.py Module</b>	<b>65</b>
17.1	A Very Brief Introduction to Classes and Objects . . . . .	65
17.1.1	Instantiating a Class With Additional Information . . . . .	66
17.2	Modules Revisited . . . . .	66
17.3	The graphics.py Module . . . . .	67
<b>18</b>	<b>When Graphics Goes Wrong!</b>	<b>69</b>
18.1	When Graphics Goes Right . . . . .	69

18.2 What Can Go Wrong? . . . . .	69
18.2.1 User Input . . . . .	70
<b>19 Exam 2 Review</b>	<b>71</b>
<b>20 Exam 2</b>	<b>71</b>
 <b>III Intermediate Python Programming</b>	 <b>73</b>
<b>21 Python Conditionals</b>	<b>75</b>
21.1 Review: Conditional Statements . . . . .	75
21.2 Python Conditionals . . . . .	75
21.2.1 Write Real Statements . . . . .	76
21.2.2 Indentation . . . . .	76
<b>22 Writing an Interactive Graphics Program</b>	<b>79</b>
22.1 Today's Activity . . . . .	79
<b>23 Python Functions</b>	<b>81</b>
23.1 What Is a Function? . . . . .	81
23.1.1 What Do Functions Do? . . . . .	82
23.2 What Makes a Function Run? . . . . .	82
23.3 How Can We Tailor a Function to Our Needs? . . . . .	82
<b>24 Tracing Python Functions</b>	<b>83</b>
<b>25 Validating User Input</b>	<b>85</b>
25.1 Introduction to Loops . . . . .	85
25.2 Iteration . . . . .	86
25.3 Syntax . . . . .	86
<b>26 Graphics Programming Activity</b>	<b>87</b>
<b>27 Finite Loops and Collections</b>	<b>89</b>
<b>28 Traffic Light Activity</b>	<b>91</b>
<b>29 Exam 3 Review</b>	<b>93</b>
<b>30 Exam 3</b>	<b>93</b>
<b>Appendices</b>	<b>97</b>
<b>A IDLE</b>	<b>97</b>
A.1 Using IDLE . . . . .	97
A.1.1 Windows . . . . .	97



A.1.2	Mac OS . . . . .	99
A.1.3	Linux . . . . .	99
<b>B</b>	<b>Python Keywords</b>	<b>101</b>
	<b>Index</b>	<b>103</b>



# List of Figures

1.1	Typical Lecture Introduction Slide . . . . .	4
2.1	A syntax error in English . . . . .	12
2.2	A semantic error in English . . . . .	12
3.1	Dates Problem Matrix . . . . .	18
7.1	Flawed Thinking About High School . . . . .	34
7.2	Less Flawed Thinking About High School . . . . .	34
7.3	Determining a Student's Major With a Nested Conditional . . . . .	35
11.1	The Hello World Program in IDLE on Linux . . . . .	47
12.1	Syntax Error: Calling a Function Without Parentheses . . . . .	52
A.1	IDLE Running on Windows 10 . . . . .	98
A.2	IDLE Running on Windows 10 with the Fira Code Font . . . . .	99
A.3	IDLE Running on Ubuntu Linux . . . . .	100



# List of Code Examples

11.1	Code Example 11.1: Hello World . . . . .	46
15.1	Code Example 15.1: Hello World With a Full Header . . . . .	62
17.1	Code Example 17.1: Testing the Graphics Module . . . . .	68
25.1	Code Example 25.1: If Statements and While Statements . . . . .	86



# List of Tables

6.1	Relational Operators in Python . . . . .	32
7.1	And, Or, and Not Truth Tables . . . . .	37
B.1	The Keywords of the Python Language . . . . .	101





# About These Notes

## Welcome Future Programmer!

Hello! Thank you and congratulations for choosing to study programming and problem solving at Nassau Community College! Computers are ubiquitous in our society, and so a knowledge of how they work is very important for the educated person in the 21st century. This course will provide a unique opportunity for you to discover how computers work, by telling them what to do!

While this course will not make you an expert programmer, learning the basics of programming – and, perhaps just as importantly, learning how programmers think – will give you the insights you need to understand what drives our economy, our culture, our technology, and virtually every other facet of our lives. Even a cursory understanding of programming and of programmers will make you a better physicist, or accountant, or elementary school teacher.

## How to Use This Document

This document is meant to introduce you to the topics that you will be covering each day in class. It is recommended that you read each day's notes **before** coming to class. The discussions you have in class are not meant to introduce you to these topics, but to **reinforce** the topics. The best strategy for succeeding in this class is to avoid surprises! Arm yourself with as much information as possible *before* attending class, and then capitalize on your new knowledge after class by *practicing* what you've learned.

## Ethical Behavior

As Stan Lee taught us through his character Spider-Man, with great power comes great responsibility. As the world's activities become increasingly electronic, and increasingly interconnected, we programmers must do our part to ensure the rights – including the right to privacy – of those whose lives our work will affect. The ethics

of programming will be discussed at various points in these notes and during the course.

To that end, we NCC Computer Science and Information Technology faculty subscribe to the [Code of Ethics](#) published by the [Association for Computing Machinery](#), and we will refer to this Code from time to time.

## How This Document Was Created

The tools used to create this document include:

- The  $\text{\LaTeX}$  document typesetting system (<https://www.latex-project.org/>)
  - The  $\text{\XeTeX}$  (<http://xetex.sourceforge.net/>) typesetting extension
- Mozilla Project's Fira Type Family (<https://github.com/mozilla/Fira>)
- VS Code (<https://code.visualstudio.com/>)
  - James Yu's LaTeX Workshop extension (<https://marketplace.visualstudio.com/items?itemName=James-Yu.latex-workshop>)
- Green Mountain Coffee Roasters Colombia Select (<http://www.gmcr.com/>)

## Acknowledgments

My sincere thanks to my colleagues in the Department of Mathematics, Computer Science, and Information Technology, who encouraged me to apply for a sabbatical, and who taught my classes in my absence, so that I could prepare this material. In particular, thanks to the members of the CSC 104 course committee, who helped guide me during some critical decision making. Thanks also to the Nassau Community College Federation of Teachers' Sabbatical Committee, for their hard work to ensure that professors like me get the opportunity to do this kind of work.

# **Part I**

## **Problem Solving**



# Day 1

## Course Introduction

If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions.  
– *Albert Einstein*

### Contents

1.1	Course Strategies . . . . .	3
1.2	Daily Preview . . . . .	4
1.3	Interview Questions . . . . .	5
1.4	Logo Exercise . . . . .	5
1.5	Homework . . . . .	5

Welcome to **CSC 104: Programming Logic and Problem Solving!** This course is intended for Information Technology majors, Computer Science majors, and anyone else who's interested in learning about computer programming and about computer programmers.

## 1.1 Course Strategies

Two personality traits that seem to be common to successful programmers are:

- Curiosity
- Perseverance

Why is that? Many computer programmers spend a lot of time solving problems that have never been solved before. This often requires thinking about an old problem in a new way, and deciding not to give up.

Also, sometimes the best result is not a complete answer, but just an answer that's incrementally better than the most recent answer. Small successes are successes, and we must accept them and celebrate them! Sometimes programmers talk about being **productively lost** – not quite sure which way to proceed, but at least having a good understanding of the problem one is in.

For these reasons, it is important that you **ask lots of questions**, during class, in your professor's office hours, or over electronic communication. Every question moves the conversation, and aids in problem solving. Even asking a "bad" question helps eliminate an avenue of inquiry that might not have led to success.

Related to this idea, it's important that you **answer every question** during class or on homework. Even if your answer is wrong or incomplete, a "bad" answer is something you can build upon.

Make sure to **review what you've learned** after every class, and make sure everything makes sense. Ask questions as soon as you have to, in order to get back on track.

Finally, make sure you understand your professor's policies. **Read the syllabus** and understand your responsibilities, and what's expected of you. Check on when things like exams are expected to happen, and add them to your calendar now.

## 1.2 Daily Preview

Each day, your professor will display a slide like this:

Figure 1.1: Typical Lecture Introduction Slide

**Day 1 Introduction**

**Problem**  
I'm new here

**Tools**  
Pen, paper

**New Tools**  
Syllabus, email, CLC

How well can you create instructions? How well can you follow instructions?

1

Understanding the day's problem, tools to be used, and new tools to be acquainted with, is a good way to get into the right mindset for the day's activities.

## 1.3 Interview Questions

It was once very popular for hiring officers to ask programmers to solve very difficult logic puzzles during job interviews. These questions are no longer used so frequently, but they were popular because, it was thought, they allowed the interviewers to see how the interview subjects think about new problems. As we work through the first part of this course, we will examine questions like this, to try to gain insight into how programmers think, and why it seems that programmers think about things in a different way than other people do.

The article that begins on Page 6 at the end of today's notes describes the situation well.

## 1.4 Logo Exercise

The best way to learn how to do anything is to do it. Several times this semester, you will be engaging in a hands-on activity, meant to help you enhance a skill that you will be relying on throughout the rest of the course. Today, you will learn just how difficult it can be to write directions and to follow them.

## 1.5 Homework

For homework, make sure you do the following things:

- Sign up for any web sites your professor asked you to sign up for, like the course management system, communications tools, etc.
- Purchase access to the zyBooks textbook
- Familiarize yourself with the location of the Computer Learning Center in B 225



## Want a job at Google? Try these brainteasers first

Google, Microsoft, and eBay are looking for engineers who can think on their feet. Here's how they find them.

By Michael Kaplan, Business 2.0 Magazine  
August 30 2007: 9:17 AM EDT

<http://money.cnn.com/2007/08/29/technology/brain.teasers.biz2/>

(Business 2.0 Magazine) – Dream of landing a coding job at an A-list tech company? It might be a good idea to prep for your interviews by pondering how many golf balls can fit inside a school bus. Or how much you would charge for washing all the windows in Seattle. Or why, exactly, manhole covers are round and not, say, square.

Seemingly random questions like these have become commonplace in Silicon Valley and other tech outposts, where companies aren't as interested in the correct answer to a tough question as they are in how a prospective employee might try to solve it. Since businesses today have to be able to react quickly to shifting market dynamics, they want more than engineers with high IQs and good college transcripts. They want people who can think on their feet.

Microsoft often gets credit for bringing so-called open-ended logic-problem screening tools into vogue in the late 1980s, when Redmond interviewers peppered job candidates with offbeat questions like How much does a 747 weigh? "We want to gauge people's creativity," says Warren Ashton, recruiting manager at Microsoft. The manhole cover problem is Ashton's personal favorite.

The most common answer, he says, is that a square manhole cover, tipped at an angle, could fall through the hole. "But some people recognize that you can roll a round manhole cover from site to site. Others figure that you save money by making it round because of tooling requirements. You want to see people taking their conclusions as far as possible."

Such questions are more relevant to a high-tech job interview than you might think. "Employers want to see if you can make an estimate in the ballpark, within an order of magnitude," says Mark Jen, a former Google employee who is now a program manager at Tagged. Coders are constantly making educated guesses rather than calculating exact answers, so a good interview should probe how well a candidate handles such estimates. That's why Amazon.com interviewers, for example, have been known to ask job candidates to guess how many gas stations there are in the United States or to ballpark that bill for washing all of Seattle's windows.

But today's interviews go beyond seat-of-the-pants estimation. Author, design consultant, and veteran coder Bruce Eckel likes to have job candidates describe a chicken using a programming language. eBay often hits candidates with a word problem that goes like this: You have five pirates, ranked from 5 to 1 in descending order. The top pirate has the right to propose how 100 gold coins should be divided among them. But the others get to vote on his plan, and if fewer than half agree with him, he gets killed. How should he allocate the gold in order to maximize his share but live to enjoy it? (Hint: One pirate ends up with 98 percent of the gold.)

But no company has taken brainteaser recruiting quite as far as Google, which famously reeled in engineers three years ago by posting complex math problems on a billboard along Highway 101 in Silicon Valley. Passing motorists were invited to submit their solutions to an undisclosed website. (The site's URL was hidden in the answer.)

If you got to the site, you were asked a second, more difficult question. If you answered that one correctly, you were invited to submit your resume. Once you got to the Googleplex for an interview, a favorite question was this: You are shrunk to the height of a nickel and your mass is proportionally reduced so as to maintain your original density. You are then thrown into an empty glass blender. The blades will start moving in 60 seconds. What do you do?

It's not just employees who have to adjust to the new screening processes. Employers are also prepping for interviews in ways they never did before. When LinkedIn founder Reid Hoffman

was searching for a new CEO earlier this year, he took an unusual approach to checking references. Having set his sights on a particular candidate, he used the LinkedIn network to find what he calls “off-balance references”: 23 former associates who were not preapproved by the candidate. Some were friends of friends – two degrees removed, in LinkedIn parlance. Some had no idea who Hoffman was or why he was calling.

The unscripted references helped to prepare the team that interviewed Hoffman’s final choice. Equally important, Hoffman got unfiltered information about a potential top-tier employee, minimizing the likelihood of getting duped. “Normally it’s a low bar for someone to give you two or three people who’ll say nice things about them,” Hoffman says. “Our way of doing it requires a bit of detective work, and you need to put a story together. But you quickly sense if a person is good or a sham.”

Getting a handle on a candidate’s people skills can be just as important – and just as tricky. At aQuantive, a digital marketing company, job applicants go through an extensive interview loop and in most cases must win the approval of everyone who met them before getting an offer. “We jokingly liken it to organ rejection,” says Kem Day, director of recruiting. “But by reaching a better decision up front, we make sure that whoever we hire will get support from within the company.”

Clearly, technical skills alone are not going to cut it in today’s high-tech environment. “It used to be that if you could spell ‘engineer,’ you got a shot,” says Beverly Principal, assistant director of Stanford University’s Career Development Center. “Now there is much more concern about employees being able to work with the company and grow to the next level.” That, and being able to survive when shrunk to the size of a nickel.

*How many golf balls can fit in a school bus?*

About 500,000, assuming the bus is 50 balls high, 50 balls wide, and 200 balls long

*You’re shrunk and trapped in a blender that will turn on in 60 seconds. What do you do?*

Some options:

1. Use the measurement marks to climb out
2. Try to unscrew the glass
3. Risk riding out the air current

*How much should you charge to wash all the windows in Seattle?*

Assuming 10,000 city blocks, 600 windows per block, five minutes per window, and a rate of \$20 per hour, about \$10 million

*Michael Kaplan is a writer in Brooklyn, N.Y. Brainteaser solutions by Mark Jen, engineer and interview junkie.*



## Day 2

# Talking Like a Programmer

It's all talk until the code runs.

– Ward Cunningham

## Contents

2.1	Interview Question . . . . .	9
2.2	Vocabulary . . . . .	10
2.2.1	What Is a Computer Program? . . . . .	10
2.2.2	What Is an Algorithm? . . . . .	10
2.2.3	What's In a Computer? . . . . .	11
2.2.4	Kinds of Programming Errors . . . . .	12
2.2.5	How the Computer Reads Our Instructions . . . . .	12
2.3	Three Fundamental Components of a Program . . . . .	13
2.3.1	Sequential Statements . . . . .	13
2.3.2	Selection Statements . . . . .	13
2.3.3	Iteration Statements . . . . .	13
2.4	Programming Ethics . . . . .	14
2.4.1	General Principles . . . . .	14
2.5	Writing and Following Directions Exercise . . . . .	14
2.6	Homework for Day 3 . . . . .	14

## 2.1 Interview Question

Your professor will give you a challenge regarding the crossing of a rope bridge. Remember – what's most important is that you make a guess, even if it isn't right. A

“bad” guess can help lead to the correct answer!

## 2.2 Vocabulary

Some people think, when they hear programmers talk, that they’re speaking a foreign language. Hopefully, today’s lesson will help familiarize you with some of the jargon we use. If you’re going to do programming, it’s important that you understand the spoken language that will be used, so that you can converse with the people you’ll be working among.

### 2.2.1 What Is a Computer Program?

A **computer program** consists of a series of instructions that we humans write, using a vocabulary that the computer can understand. When the computer *executes*, or runs, our instructions, it will do so exactly as they are written, and so it’s important that we write these instructions using the right language, in the right order, and without any ambiguity.

A person who is in the process of creating a program is said to be engaged in **computer programming**. But programming isn’t just the act of typing stuff on a keyboard; it also involves **testing** the program to see whether it works, and then **debugging**, or removing the problems we’ve programmed in.

### 2.2.2 What Is an Algorithm?

This leaves us with the question of what to type in. How do we know which instructions to give to the computer?

The process of programming starts with designing an algorithm. An **algorithm** is a set of instructions that can be used repeatedly to solve a problem. You probably have some algorithms you follow every day, like preparing breakfast or walking to class from the parking lot. You could probably sit at home and describe to another student over the phone how to walk to the B building from where you normally park. Some important characteristics of algorithms are that they’re **repeatable** and **describable** to another person.

A programmer, then, describes an algorithm to a computer, by writing instructions in the computer’s language.

### 2.2.3 What's In a Computer?

Most modern computers, whether on your desk or in your pocket, have these components:

#### Central Processing Unit

The **central processing unit**, or **CPU**, is the part of the computer that performs all of the instructions we write. It keeps track of what instruction needs to be performed next, and makes sure the right things show up on the screen and get stored in memory. CPUs are frequently measured by how many billions of operations they can perform in a second, using the unit of measurement *gigahertz* (GHz).

#### Memory

The computer's **memory**, also called *primary storage*, stores programs until they're ready to be run, and it also stores data and information that programs need while they run. When you learn about *variables* later, you will learn that they get stored here.

We measure the capacity of a computer's memory in *gigabytes*, or how many billions of bytes it can store. In a typical desktop computer, the memory used by programs (RAM, or *random access memory*) only works when there's a constant supply of electricity; if the computer loses power, the memory will be emptied.

#### Drives, Disks, and Long-Term Storage

There are many kinds of *secondary storage* available for various kinds of computers. They are frequently referred to as **hard drives** or *disks*, although these terms don't apply to some of the newest choices. Whether they use old-fashioned spinning platters or not, however, they are used to store lots of data for a long time, including when the computer is off. Hard drives are used to store our programs when they're not in use, and our files, like term papers, saved game progress, and songs.

#### Input and Output

Keyboards, screens, mice, speakers, and similar devices allow us to **input** information into the computer, and allow the computer to **output** information back to us. Computers can work just fine without these devices, but not in an interactive way.

### 2.2.4 Kinds of Programming Errors

There are three broad categories of errors that can exist in a computer program. The first is an error in the structure of our instructions, and the second is an error in the logic of our instructions. (The third is an error in the execution of our instructions, but we'll talk about these kinds of errors later.)

A **syntax error** is an error that arises when we programmers type something that does not follow the structural rules of the programming language in which we're writing. Consider a similar error in a natural language. What's wrong with this English sentence?

Figure 2.1: A syntax error in English

***This sentence no verb.***

The problem there is that we broke the rules of English – specifically, the rule that says a sentence has to have a verb.

Now consider this sentence:

Figure 2.2: A semantic error in English

***My dog drove me to school today.***

The structure of this sentence is valid. A grammarian would have no problem with this sentence. A logician, however, would bristle at this, because it doesn't make any sense. The sentence in Figure 2.2 doesn't contain any syntax errors, but it does contain an error. Specifically, this sentence has a **semantic error**, which is an error in the logic or meaning of a statement.

### 2.2.5 How the Computer Reads Our Instructions

Certainly we humans don't communicate with each other using computer programming languages, and so when we write programs, we're performing a sort of translation, or interpretation, of our algorithm into that language. You may be surprised, however, to learn that even a programming language is not a computer's native language. The only language the computer natively understands is numerical in nature – instruction 1 might mean “add these two numbers,” and instruction 2 might mean “compare these two numbers, and store the result over here.” Therefore, we need a program to translate our computer programming language instructions –

which we call **source code** – into machine language instructions. There are two ways this can happen.

Some languages utilize a program called a **compiler**, which examines all of our source code at once. If all of the source code is free of syntax errors, the compiler will then translate the entire program to machine language code. The code that comes from the compiler is saved on disk, and can be run multiple times without having to be compiled again. Other languages' source code is fed into an **interpreter**, which attempts to execute each instruction of a program separately. An interpreter can run some of a program, even if there's a syntax error somewhere in the middle; however, once a syntax error is encountered, the interpreter will immediately exit.

The language we will be programming in this semester, called Python, is a compiled language, but the resulting code is not machine code. Instead, the Python compiler creates something called **bytecode**, which then must be run by an interpreter.

## 2.3 Three Fundamental Components of a Program

There are three basic kinds of instructions that we programmers can write when implementing our algorithms in computer code.

### 2.3.1 Sequential Statements

It's important for some instructions to take place in a certain order. When you're making breakfast, you can't put your knife in the preserves jar if you haven't taken the lid off first. **Sequential statements** must be typed in in the proper order when necessary.

### 2.3.2 Selection Statements

Did you plug in your mobile phone when you got to class today? Why? Is there a certain threshold of battery percentage at which you worry about it lasting through class? We can write certain instructions for the computer to select what to do, based upon a condition. **Selection statements** (also known as **conditional statements**) allow us to allow the computer to make a decision. We'll explore those more in a few weeks.

### 2.3.3 Iteration Statements

How was parking today? Did you have to drive in circles around a section of the parking lot? Any repeated action like that is called *iteration* by programmers, and



so an **iteration statement**, or *loop*, is the kind of instruction we write when we want some task to happen over and over again.

## 2.4 Programming Ethics

From time to time in this course, we will discuss the responsibilities we programmers have to the people whose lives our work might affect. The [Code of Ethics](#) published by the [Association for Computing Machinery](#) is a resource that is widely accepted throughout the computing industry, and one to which all of us that teach Computer Science and Information Technology here at Nassau subscribe.

### 2.4.1 General Principles

The first part of the Code of Ethics lists a programmer's General Principles. Our first principle is this:

Contribute to society and to human well-being, acknowledging that all people are stakeholders in computing. [[direct link](#)]

This principle affirms our responsibility to use our talents for the benefit of society. It is not only easy, but far too common, for people, organizations, and corporations to forget this principle and, either through mistakes or through overt acts, create software and computer systems that do not benefit society, and potentially even cause harm to people or computers. Computers – and, by extension, programmers – have changed the world for the better, but we must always strive to maintain the trust that society has placed in us.

## 2.5 Writing and Following Directions Exercise

Your instructor is going to ask you and a partner to write some instructions, but with some restrictions. Then we will see how you did.

## 2.6 Homework for Day 3

Ted and Ken, and their wives Allyson and Janie each have a favorite sport. The sports they enjoy are running, swimming, biking, and golf. Based upon these clues, determine whose favorite sport is which, and submit the answer for homework

before Day 3 begins. Your instructor will give you instructions for submitting your answer.

1. Ted hates golf.
2. Ken wouldn't run around the block if he didn't have to, and neither would his wife.
3. Each woman's sport is featured in a triathlon.
4. Allyson bought her husband a new bicycle for his birthday, to use in his favorite sport.



## Day 3

# Matrix Logic

Unfortunately, no one can be told what the Matrix is. You have to see it for yourself.  
– Morpheus

### Contents

3.1	What Is Matrix Logic? . . . . .	17
3.2	Example Matrix Problem . . . . .	17
3.3	Matrix Logic Exercises and Homework . . . . .	21

## 3.1 What Is Matrix Logic?

It can be hard to know how to solve a problem that has multiple elements and multiple possibilities. Part of the transition into thinking like a programmer involves the ability to organize information and eliminate impossibilities in a systematic way. A **matrix** helps us set up a one-to-one correspondence among the elements in a problem.

## 3.2 Example Matrix Problem

**The problem statement:** Three friends, Sarah, Natalie, and Charlotte, all had dates on Saturday night, with Nick, Matt, and Ben. One couple went to the movies, one went to the park, and one went out to dinner. Based on these text messages, see if you can determine who went out with whom, and where they went.

**The clues:**

5:45

Can't believe how cold it is and the grass is all wet, yuck. It's okay though, Matt gave me his coat, I think he really likes me.

8:34

Hey guys, hope you're having a better time than I am. Ben is such a bore, I don't think I'll be doing this again. xoxoxo love, Sarah oxoxox

9:24

Wow, this movie is fantastic and he's just as into it as I am. I don't think it'd ever be as fun with a geek like Matt (no offence). I don't ever want this to end. Good luck on your dates, love Natalie

**The strategy:** Did Sarah go out with Ben? Did Natalie go to the park? We need a system of solving this problem that helps us eliminate the combinations that couldn't have happened, so that we leave only the one solution that makes sense.

For a problem like this, the best strategy is to create a **matrix** like this:

Figure 3.1: Dates Problem Matrix

	Nick	Matt	Ben	Movies	Park	Dinner
Sarah						
Natalie						
Charlotte						
Movies						
Park						
Dinner						

Some of the clues in a problem like this help us match two pieces of information. We will place a capital letter **O** where we have a match. Read the first text message again. We don't know who sent it, but we know she's in the park with Matt, so we can update the matrix like this:

	Nick	Matt	Ben	Movies	Park	Dinner
Sarah						
Natalie						
Charlotte						
Movies						
Park		O				
Dinner						

Once we've placed the O, it becomes clear that neither Nick nor Ben is at the park, and that Matt is neither at the movies nor at dinner. We can mark these impossibilities with an X, like this:

	Nick	Matt	Ben	Movies	Park	Dinner
Sarah						
Natalie						
Charlotte						
Movies		X				
Park	X	O	X			
Dinner		X				

The second text message tells us that Sarah is out with Ben. This means that Sarah is not with Nick or Matt, and that Ben is not with Natalie or Charlotte:

	Nick	Matt	Ben	Movies	Park	Dinner
Sarah	X	X	O			
Natalie			X			
Charlotte			X			
Movies		X				
Park	X	O	X			
Dinner		X				

But there's more information to glean from this clue. Since we already know that Ben is not at the park, then neither is Sarah, and so we can eliminate that possibility as well:

	Nick	Matt	Ben	Movies	Park	Dinner
Sarah	X	X	O		X	
Natalie			X			
Charlotte			X			
Movies		X				
Park	X	O	X			
Dinner		X				

From the third text message, we learn that Natalie is at the movies. Since Ben is out with Sarah, and Matt is at the park, Natalie must be out with Nick.

	Nick	Matt	Ben	Movies	Park	Dinner
Sarah	X	X	O		X	
Natalie	O	X	X	O	X	X
Charlotte	X		X			
Movies	O	X				
Park	X	O	X			
Dinner		X				

We are just about done with this problem at this point. Notice that in the lower left, once the X's are filled at Nick/Dinner and at Ben/Movies, it's clear that Ben is at Dinner, which means that Sarah is at dinner. Similarly, with two X's under Park in the top right, it must be Charlotte at the park, which means that that first text message, about Matt, was from her.

	Nick	Matt	Ben	Movies	Park	Dinner
Sarah	X	X	O	X	X	O
Natalie	O	X	X	O	X	X
Charlotte	X	O	X	X	O	X
Movies	O	X	X			
Park	X	O	X			
Dinner	X	X	O			

And so: Sarah and Ben are at dinner, Natalie and Nick are at the movies, and Charlotte and Matt are at the park.

### 3.3 Matrix Logic Exercises and Homework

You will have the opportunity to solve a couple of problems like this in class. Answers to the last one should be submitted for homework.





## Day 4

# Systematic Lists

It's not that I'm so smart, it's just that I stay with problems longer.

– Albert Einstein

### Contents

4.1	Interview Question . . . . .	23
4.2	Systematic Lists . . . . .	23
4.2.1	Why Use Systematic Lists? . . . . .	24
4.2.2	How Do We Create a Systematic List? . . . . .	24
4.2.3	Creating a Systematic List . . . . .	24
4.3	Characteristics of a Systematic List . . . . .	27
4.4	Creating Systematic Lists . . . . .	27

## 4.1 Interview Question

Day 4 starts with an interview question about the competitors in a tournament. See if you can answer the question without help. If not, see if a nearby classmate can give you a hint.

## 4.2 Systematic Lists

A **systematic list** is a list of related items that are listed in a predetermined order. The ability to create such a list is a skill that programmers use in solving a certain kind of problem.

### 4.2.1 Why Use Systematic Lists?

Creating a systematic list is a great way to explore *combinations* of items or values. Here's an example: Suppose that Allison, Becky, and Charlotte are running a race. What are all the race results that can happen? It's possible for Becky to win, followed by Charlotte and then Allison, but it's also possible for Allison to win, followed by Charlotte and then Becky.

It seems like human nature for us to discuss these possible results in this way, by discussing each result as we think of it. However, there are two potential problems in this scenario:

- There is no way to be sure that we haven't skipped any results
- There is no way to know that we completed the task

Using a systematic list helps solve both of those problems.

### 4.2.2 How Do We Create a Systematic List?

The best way to avoid these problems is to attack each situation the same way. We will develop an *algorithm*, and follow it. A successful algorithm will not only work every time, but finish as soon as it has to.

An **algorithm** is, according to Merriam-Webster, "a step-by-step procedure for solving a problem or accomplishing some end." A good algorithm should be *describable*, *repeatable*, and *verifiable* – that is, we should be able to talk about it, use it, and know it works.

The most successful algorithm for creating a systematic list relies heavily upon the *order* of the items to be listed. Notice, in our previous example, that the race runners have a natural ordering criteria – we can alphabetize their names. A statement like "Becky's name comes before Charlotte's" makes sense to us. We will use this feature of people's names to our advantage.

Our systematic list algorithm involves repeating this step over and over:

- Add the smallest unused value in the left-most available position
- Only use the next value when all combinations utilizing the present value have been enumerated

### 4.2.3 Creating a Systematic List

We will fill in this table with as many rows as we need:

First Place	Second Place	Third Place
<i>result</i>	<i>result</i>	<i>result</i>

Using our race running example, the “first” name we have available is Allison, and so we will start by enumerating all of the race results in which she is the winner. Let’s place Allison in the left-most position.

First Place	Second Place	Third Place
Allison	<i>result</i>	<i>result</i>

Once we’ve done that, as the algorithm says, we will add the smallest unused value in the left-most available position. The “smallest” name we haven’t used yet is Becky’s, and the left-most available position is next to Allison.

First Place	Second Place	Third Place
Allison	Becky	<i>result</i>

There is one more spot in this row to fill, and one more name – the “largest” or latest alphabetically – to add.

First Place	Second Place	Third Place
Allison	Becky	Charlotte

Because there are no other names to replace Charlotte’s with, we will consider ourselves done with all of the combinations that start with *Allison and Becky*.

Note, however, that there are more possible results that start with Allison. Let’s backtrack to that scenario:

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	<i>result</i>	<i>result</i>

Remember what our algorithm tells us to do: *Add the smallest unused value in the left-most available position*. The smallest name we haven’t used for second place yet is Charlotte, so let’s add her:

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	<i>result</i>

In this row, we have used Allison’s name and we have used Charlotte’s name, but we have not used Becky’s name. The algorithm tells us to *Add the smallest unused value in the left-most available position*. That means add Becky to the third column:

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	Becky

Since we’ve now used every runner’s name in this row, we can conclude that there are no more combinations that start with *Allison and Charlotte*.

Let’s backtrack once again:

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	Becky
Allison	<i>result</i>	<i>result</i>

Are there any more combinations that start with just *Allison*? In other words, if Allison wins the race, are there any other orders in which the runners can finish?

The algorithm says *Add the smallest unused value in the left-most available position*. But here in Column 2, there are no more unused values. The next part of the algorithm says *Only use the next value when all combinations utilizing the present value have been enumerated*. Since we have now enumerated all of the conditions utilizing the present value – that is, we’ve listed all the results possible when Allison wins – we are now free to use the next value.

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	Becky
Allison	<i>result</i>	<i>result</i>

In this case, the next value is the next smallest value in Column 1, meaning it’s time to see what can happen if Becky wins the race.

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	Becky
Becky	<i>result</i>	<i>result</i>

At this step, we ask once again what the smallest remaining value is. We haven’t used Allison’s name in this row yet, so we put her name into Column 2 now.

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	Becky
Becky	Allison	<i>result</i>

That leaves Charlotte to finish third:

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	Becky
Becky	Allison	Charlotte

As before, there is another name available to put right after Becky’s:

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	Becky
Becky	Allison	Charlotte
Becky	Charlotte	Allison

There is only one winner left to explore, and we complete the systematic list by completing the algorithm:

First Place	Second Place	Third Place
Allison	Becky	Charlotte
Allison	Charlotte	Becky
Becky	Allison	Charlotte
Becky	Charlotte	Allison
Charlotte	Allison	Becky
Charlotte	Becky	Allison

### 4.3 Characteristics of a Systematic List

When you are asked to produce a systematic list of all the outcomes of this race, it is not enough to just list all of the outcomes; they must be listed in the proper order. You must create a systematic list, not just a list.

The system in this list lies in the ordering. First, notice that the values in the first column never decrease. All of the outcomes in which Allison wins are listed before any of the others, and all of the outcomes in which Becky wins are listed before any in which Charlotte wins.

Then, when you consider all of the outcomes that have the same first-column value, notice that the second-column values are also written in non-decreasing order – Becky then Charlotte, Allison then Charlotte, and Allison then Becky.

### 4.4 Creating Systematic Lists

Follow your instructor's lead in creating some systematic lists in class.



## Day 5

# Binary Numbers

There are 10 kinds of people in this world – those who understand binary numbers, and those who don't.  
– A progammer's t-shirt

### Contents

5.1	Review Exercise . . . . .	29
5.2	What Is a Number System? . . . . .	29
5.3	Binary Numbers . . . . .	30

## 5.1 Review Exercise

Your instructor will ask you to create a systematic list to help you review what you learned in Day 4, and also to help introduce today's topic.

## 5.2 What Is a Number System?

A **number system** is a series of symbols that we use to represent numeric values. We are used to using **decimal numbers** in our daily lives. This number system, as indicated by the Greek root *deci*, meaning “ten,” has ten digits – namely, 0 through 9. With those ten digits, we can describe any number we're thinking of.

If we create a systematic list of two-digit combinations of the digits 0 through 9, what emerges is a list of numbers between 0 and 99, in numeric order: 00, 01, 02, ..., 09, 10, 11, 12, ..., 39, 40, 41, ..., 98, 99.



Now consider a longer number, like 3284. Each of those digits has a *place name* – there’s a ones place, and a tens place, and so forth. Why are those the names of the places? Because that number is the same as *three thousands, two hundreds, eight tens, and four ones*.

But why? Well, it is no coincidence that each of those place names is equal to a power of ten. Notice:

$$\begin{aligned} 3284 &= 3 \cdot 1000 + 2 \cdot 100 + 8 \cdot 10 + 4 \cdot 1 \\ &= 3 \cdot 10^3 + 2 \cdot 10^2 + 8 \cdot 10^1 + 4 \cdot 10^0 \end{aligned}$$

Notice that for each of those exponents, the base is ten – hence, this number was written in *base ten*.

## 5.3 Binary Numbers

It might surprise you to learn that programmers sometimes use a number system other than base ten when discussing what happens inside a computer.

At its most basic hardware level, all a computer can recognize is things like whether a voltage is above or below a certain threshold, or whether a switch is open or closed, or whether a chunk of metal is aligned magnetically north or south. All of these situations – high or low, open or closed, yes or no – represent **binary** states. (The Greek root *bi* shows up in other words you recognize, like *bicycle* and *bilingual*, and it means “two.”)

Due to this tendency of computer components to exist in one of two binary states, we use **binary numbers** to represent these states.

Just like decimal numbers, each digit in binary numbers represents a base and an exponent. With these numbers, however, the base for each exponent is two – hence, *base two* numbers. The rightmost column in a binary number represents the  $2^0$  place, or the *ones place*; the middle column is the  $2^1$  or *twos place*; and the left column is the  $2^2$ , or *fours place*. Therefore, starting from the top, those three-binary-digit sequences represent the decimal numbers 0 through 7. By the time you are done with the activities in Day 5, you should be able to convert any decimal number within a certain range to binary, and vice versa.

## Day 6

# Introduction to Conditionals

If you don't know anything about computers, just remember that they are machines that do exactly what you tell them but often surprise you in the result.  
– Richard Dawkins

## Contents

6.1	Decisions . . . . .	31
6.2	Conditions and Boolean Values . . . . .	31

## 6.1 Decisions

Life is full of decisions. Some decisions, like *should I change my major to Computer Science?* are very important, whereas others, like *should we go to Wendy's or Taco Bell for lunch?* only seem important at the time.

The earliest computer programmers faced decisions as well. Should this employee be paid a straight wage, or overtime? Is this user's input valid? Computers were designed to be good at handling decision making like this.

## 6.2 Conditions and Boolean Values

A **condition** is something that is either true or false. *It is cold out* is a condition – you can determine for yourself as you read this whether that statement is true or false right now. In a few hours, or a few months, it is possible for this condition to have the opposite value.

As we start to discuss how to write computer programs, the conditions with which we will deal most frequently will involve *comparisons*. We will learn how to ask questions like “Are you older than 21 years old?” and how to perform different tasks based on the possible outcomes.

English mathematician George Boole is usually credited with first formalizing the way we write about conditions and comparisons. His system came to be known as *Boolean logic*. Similarly, the values `False` and `True` – the only two values a condition in Boolean logic can have – are referred to as *Boolean values*. `False` and `True` are the only two Boolean values.

Today’s lesson will focus on the use of **relational operators**, which help us create a condition that examines the relationship between two values. These operators help us talk about ideas like “this number is less than that number,” or “this number is greater than or equal to that number.”

In math class, you may have encountered statements like  $x \geq y$ , which means “ $x$  is greater than or equal to  $y$ .” This is difficult to type, however, because there is no  $\geq$  key on your keyboard. Every programming language defines a set of operators that programmers must type to express these otherwise-untypeable ideas. Table 6.1 presents the relational operators you will need to know when we start programming in Python.

Table 6.1: Relational Operators in Python

Mathematical Symbol	Meaning	Python Code
$<$	Less Than	<code>&lt;</code>
$\leq$	Less Than or Equal	<code>&lt;=</code>
$>$	Greater Than	<code>&gt;</code>
$\geq$	Greater Than or Equal	<code>&gt;=</code>
$=$	Equal	<code>==</code>
$\neq$	Not Equal	<code>!=</code>

As you work through the examples today and over the next few days of class, make sure you use the Python operators, not the mathematical operators. You will use these operators today while you learn the format for a **conditional statement** and how to create your own. In later lessons you will write increasingly complex conditional statements.

While writing and testing conditional statements, make sure you pay attention to the idea of a **possible value** to be evaluated in a condition (this is the number you will compare to the *present state* of something), and make sure you understand how to determine the possible **outcomes** of a conditional statement.

## Day 7

# Nested Conditionals and Complex Conditionals

The most important property of a program is whether it accomplishes the intention of its user.  
– C. A. R. Hoare

## Contents

7.1	Nested Conditionals . . . . .	33
7.2	Complex Conditionals . . . . .	36
7.2.1	Boolean Operators . . . . .	36
7.3	Writing Conditional Statements . . . . .	37
7.3.1	The Drinking Water Example . . . . .	37
7.4	A Note About Grading . . . . .	38

## 7.1 Nested Conditionals

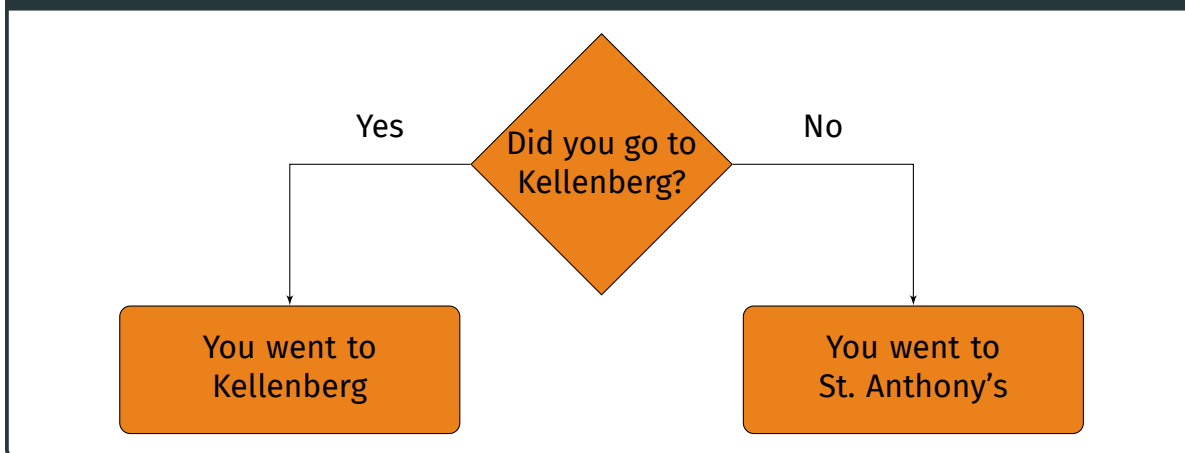
We learned in Day 6 that a condition can only have one of two values, True or False, and therefore a *conditional statement* can only have two possible *outcomes*. These ideas make conditional statements very useful for certain problems in life, but there are situations that need more complex tools than these.

Consider, for example, running into an old middle school classmate in town. You remember each other, and realize quickly that you haven't seen each other since middle school. Your friend says, "So what happened to you? Did you go to Kellenberg or something?" If you didn't, then you might be surprised if your friend follows up

with “Oh, I guess you went to St. Anthony’s.” Even if you *did* go to St. Anthony’s, that statement illustrates quite a gap in your friend’s logical reasoning skills.

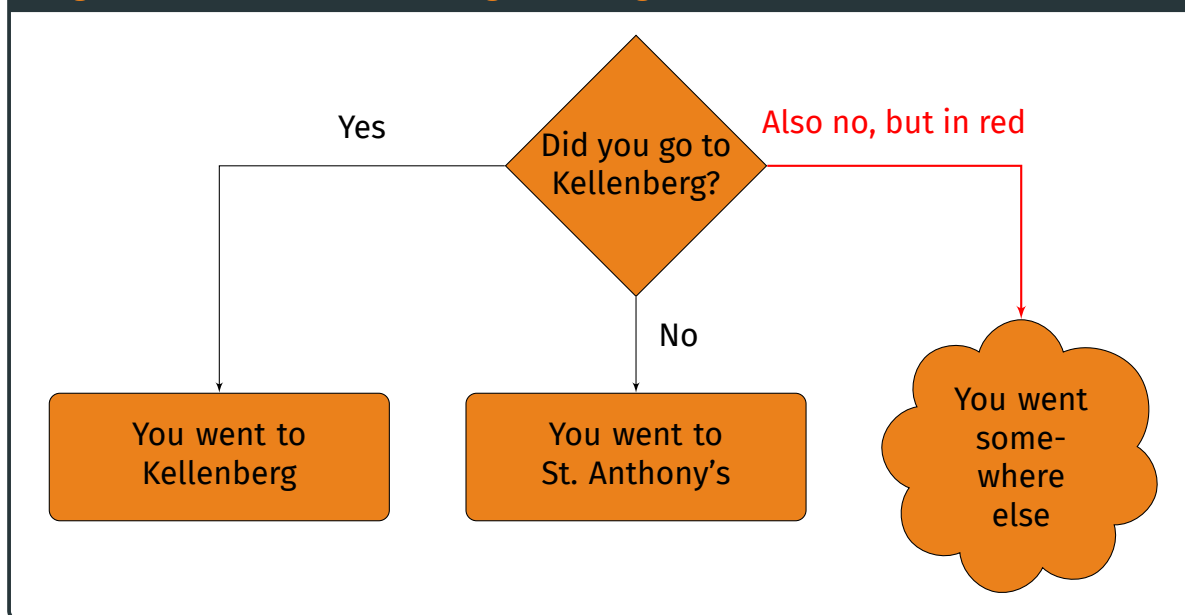
The problem is that the conditional statement that’s implied here has more *outcomes* than your friend has considered. Her thinking appears to work like Figure 7.1.

Figure 7.1: Flawed Thinking About High School



Of course, your friend should be thinking like Figure 7.2 instead.

Figure 7.2: Less Flawed Thinking About High School



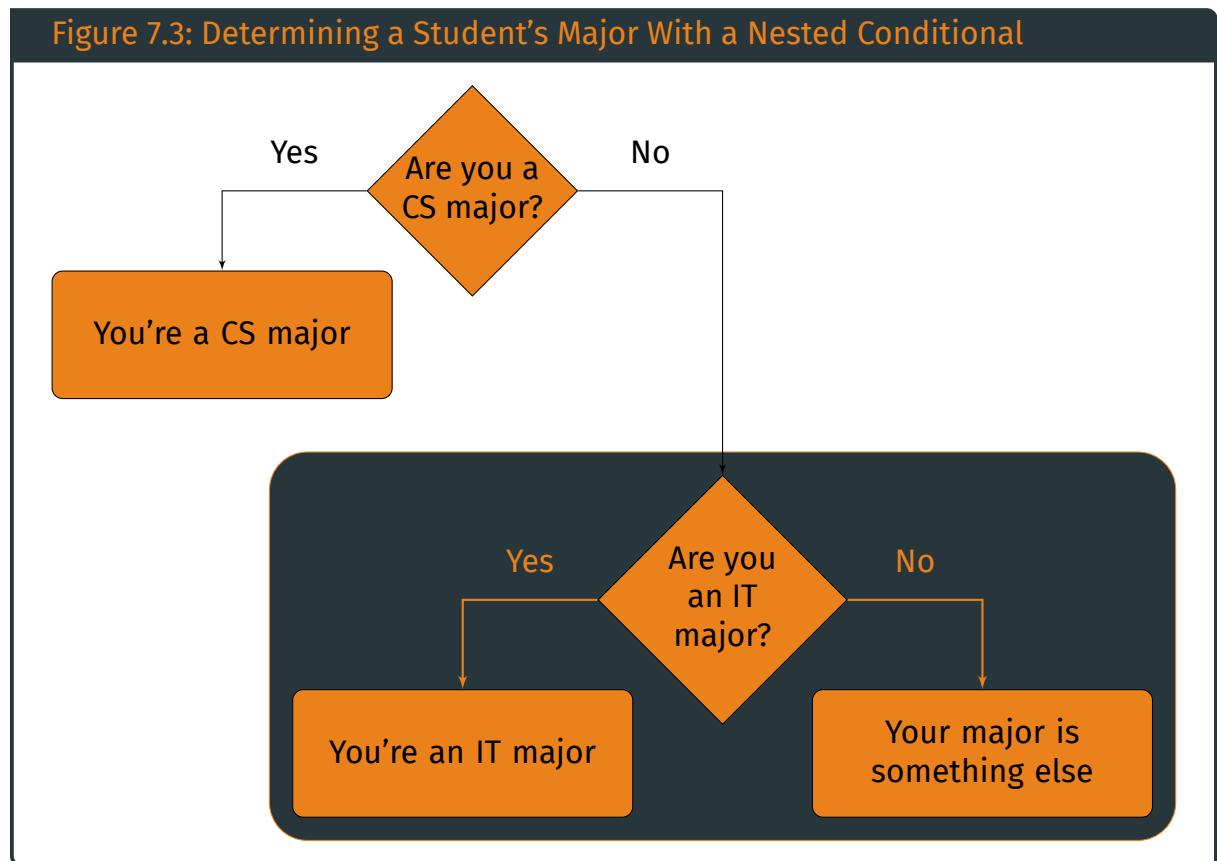
This is problematic, too. We haven’t learned about a conditional with three or more outcomes. There’s a simple reason for that – *there aren’t any*. The whole idea of a condition is that it’s either False or True. There is no third Boolean value.

Since a single conditional statement can only have two outcomes, what’s needed in a scenario like this is multiple conditional statements. Let’s consider another

example: you and your classmates all fall into one of three categories: you're either a Computer Science major, an Information Technology major, or your majoring in something else. If you ask one of your classmates "Are you a Computer Science major?", the two outcomes are:

- **Yes:** Now you know the student's major
- **No:** You don't know what the student's major is, but you know it's either IT or something else

If your classmate's answer is "Yes", you're done with the conversation. If, however, the answer is "No", then there are still two possible outcomes. Fortunately, you know how to deal with two possible outcomes – use a conditional statement! Consider Figure 7.3:



The shaded box contains a **nested conditional**. After asking the first question (is major equal to Computer Science), if the answer is no, you have to *ask another question*. In this scenario, where there are three outcomes, two questions are necessary to arrive at the right answer. In general, you must ask  $n - 1$  questions when choosing from among  $n$  outcomes.

Creating a **decision tree** like this is a good way to start thinking about how to write a nested conditional statement. Once the picture is done, you can just read your

decision tree from top to bottom and translate it into code. Note that Python accomplishes nesting conditionals like this with the **elif** clause. We will be using **elif** in our examples today and in the next few classes.

Here's the conditional statement that matches this decision tree:

```
if major = CS:
    "You're a CS major"
elif major = IT:
    "You're an IT major"
else:
    "Your major is something else"
```

## 7.2 Complex Conditionals

What will make you attend the next class meeting of CSC 104? A couple of things have to be true: it has to be a *day* on which the class meets, and it has to be a *time* when the class is meeting. If your class meets at 2:00 pm on Wednesday, then showing up at 9:30 on Wednesday isn't going to help you, and neither is arriving at 2:00 pm on Thursday.

Then again, sometimes only one thing must be true for an outcome to occur. If you're thirsty, you will drink some water. But you will also drink some water if you have to take a pill. You don't wait until both conditions are true before taking a drink of water; just one condition being true is sufficient.

Both of these scenarios can be handled using a **complex conditional**. These scenarios are similar to those from Day 6 in that there are only one or two outcomes (and therefore no nesting is necessary), but they're different in that many individual conditions go into determining which outcome happens. In these cases, we must rely on a new set of operators.

### 7.2.1 Boolean Operators

**Boolean operators** allow us to create a condition by combining other conditions together. You know these operators as "and", "or", and "not". You may also recognize them in symbol form from your high school symbolic logic section:  $\wedge$ ,  $\vee$ , and  $\neg$ .

Recall your high school truth tables, as in Table 7.1:

Table 7.1: And, Or, and Not Truth Tables

$x$	$y$	$x \wedge y$	$x \vee y$	$\neg x$
F	F	F	F	T
F	T	F	T	T
T	F	F	T	F
T	T	T	T	F

$x \wedge y$  (“ $x$  and  $y$ ”) is only true if both  $x$  and  $y$  are true; otherwise it’s false.  $x \vee y$  (“ $x$  or  $y$ ”) is only false if both  $x$  and  $y$  are false; otherwise it’s true.

Thankfully, Python does not require us to type the  $\wedge$ ,  $\vee$ , and  $\neg$  symbols (which is good, because they aren’t on the keyboard); we can just use **and**, **or**, and **not**.

With those operators, we can say things like “If it’s Wednesday **and** it’s 2:00 pm, then I will go to CSC 104”, or “If I’m thirsty **or** I need to take a pill, then I will drink water.”

## 7.3 Writing Conditional Statements

When you are asked to create a conditional statement from scratch, you should first consider your outcomes.

- Is there **one outcome**? All you need is a *simple if statement*.
- Are there **two outcomes**? Write an *if-else statement*.
- Are there **more than two outcomes**? You will need a *nested conditional statement*.

Once you’ve determined what kind of statement to write, then choose a first question to ask. The question you choose should map answers to outcomes. In other words, one of the answers to the question should always send you down one side of the flowchart, and the other answer should always send you down the other side. For the high school scenario, the question “Did you go to Kellenberg?” satisfies this criteria. A “yes” always finishes our task, and a “no” always leads us to ask more questions.

Note that these questions will sometimes actually consist of more than one condition, as explained in the example that follows.

### 7.3.1 The Drinking Water Example

Consider the scenario from before where we’re trying to determine whether to drink water or not. There is only one outcome – “drink water” or do nothing – and so we want to write one simple if statement. But there are two conditions to check on, and so they must be combined in a complex conditional statement.



You will be prompted to write several conditional statements during class today. Make sure to ask for help or clarification as soon as you need to.

## 7.4 A Note About Grading

When you are graded on writing conditional statements, part of your grade will be based on the *efficiency* of your statement – in other words, you will lose points if you ask more questions than you have to.

Avoid these common mistakes, that can cost you points on homework and exams:

- Don't ask a question that you already know the answer to. Consider the exercise you did in class on Day 6 that results in either "pass" or "fail" based on the student's average. Resist the temptation to do this:

```
if average >= 70:  
    "pass"  
elif average < 70:  
    "fail"
```

Once your program reaches the `elif`, it already knows that the answer to that question is going to be true. If the answer were false, this condition wouldn't be getting evaluated, because "pass" would have already been the outcome! This conditional leads to the right outcomes, but it does so by asking one more question than necessary. That's inefficient, and it will affect your grade.

- Don't take two questions to ask one question. This conditional statement "works":

```
if day = Wednesday:  
    if time = 2:00 pm:  
        ``go to CSC 104''
```

With only one outcome, however, only one condition needs to be evaluated, and so the two conditions above need to be combined into one complex condition.

## Day 8

# Debugging Conditionals

A good programmer is someone who always looks both ways before crossing a one-way street.

– Doug Linder

## Contents

8.1 Debugging Conditionals . . . . .	39
--------------------------------------	----

## 8.1 Debugging Conditionals

Virtually every program you ever write will, at some point in its development, be wrong. Learning how to find and fix mistakes in code is at least as important a skill to develop as learning how to write code in the first place. The exercises you will work on in class today will help you discover what these skills are, and how much work you have ahead of you in developing them.

In general, a good debugger is someone who can accomplish the following:

- **Understand what the program should do.** It seems obvious, but sometimes we can get so mired down in one little detail of a program that we can forget what it is we're trying to accomplish. Identifying a bug begins with *articulating the correct behavior*. If you can't tell someone what the program is supposed to do, do you really know whether it's doing it?
- **Understand what the program says it will do.** It is easy for to us read something – a computer program, a term paper, a blog post – and assume it says something after reading very little of it. Ask your psychology professor how our brains perform *chunking*, and how this can lead to information processing

errors. You may not have even noticed the typo in the first sentence of this bulleted item. Don't read what you think is there; read what's there.

- **Be humble.** Don't assume that it works just because you wrote it.
- **Rest often.** Every problem becomes temporarily insurmountable at some point. If you can't fix an error, let your brain, your eyes, and your body relax for a little bit, and then come back to the problem refreshed. Resist the temptation to keep pushing through when you have nothing left, or to assume that taking breaks is for the weak. Most bugs get found and fixed by fresh heads.

## Day 9

### Exam 1 Review

Your instructor will have prepared some materials for you to review in class today. Make sure to ask questions!

## Day 10

### Exam 1

Good luck! Make sure you have installed the Python programming environment on your computer before the next class!



# **Part II**

## **Programming in Python**



# Day 11

## Introduction to Python

Talk is cheap. Show me the code.  
– *Linus Torvalds*

### Contents

11.1 An Example Program . . . . .	46
11.2 How to Write and Run the Hello World Program . . . . .	46
11.3 Starting to Program in Python . . . . .	47

### Important Note: Read the Textbook

Now that we have begun the programming phase of the course, it is *very important* that you add reading the textbook to your list of pre-class preparation activities. From here on out, these notes will highlight the topics to be covered, but the textbook and its associated activities will provide the most complete information and preparation for class.

Please note that the textbook is *interactive*. It is not meant to just be read; you will be clicking on things, and eventually even typing in code, while you work through the chapters. Frequently, your homework grade will depend at least in part on completing these interactive activities.

Programming is similar to many other pursuits in that you will only get better with practice. The textbook is meant to give you these opportunities. Make sure to take advantage of them.



## 11.1 An Example Program

Let's look at what Python source code looks like. Here's a simple program to get you started.

### Code Example 11.1: Hello World

```
1 print("Hello world!")
```

It is a tradition in Computer Science, when learning a new programming language for the first time, to learn how to write a program that displays the message “Hello world!” to the screen. The Code Listing you see above in Listing 11.1 does just that.

Many people are surprised to learn that we can write a functional Python program in just one line of code, but that's exactly what we have here. Python was designed to be simple to learn, and simple to use.

## 11.2 How to Write and Run the Hello World Program

Python is not only a simple programming language to learn, it also comes with some very easy-to-use tools for programming.

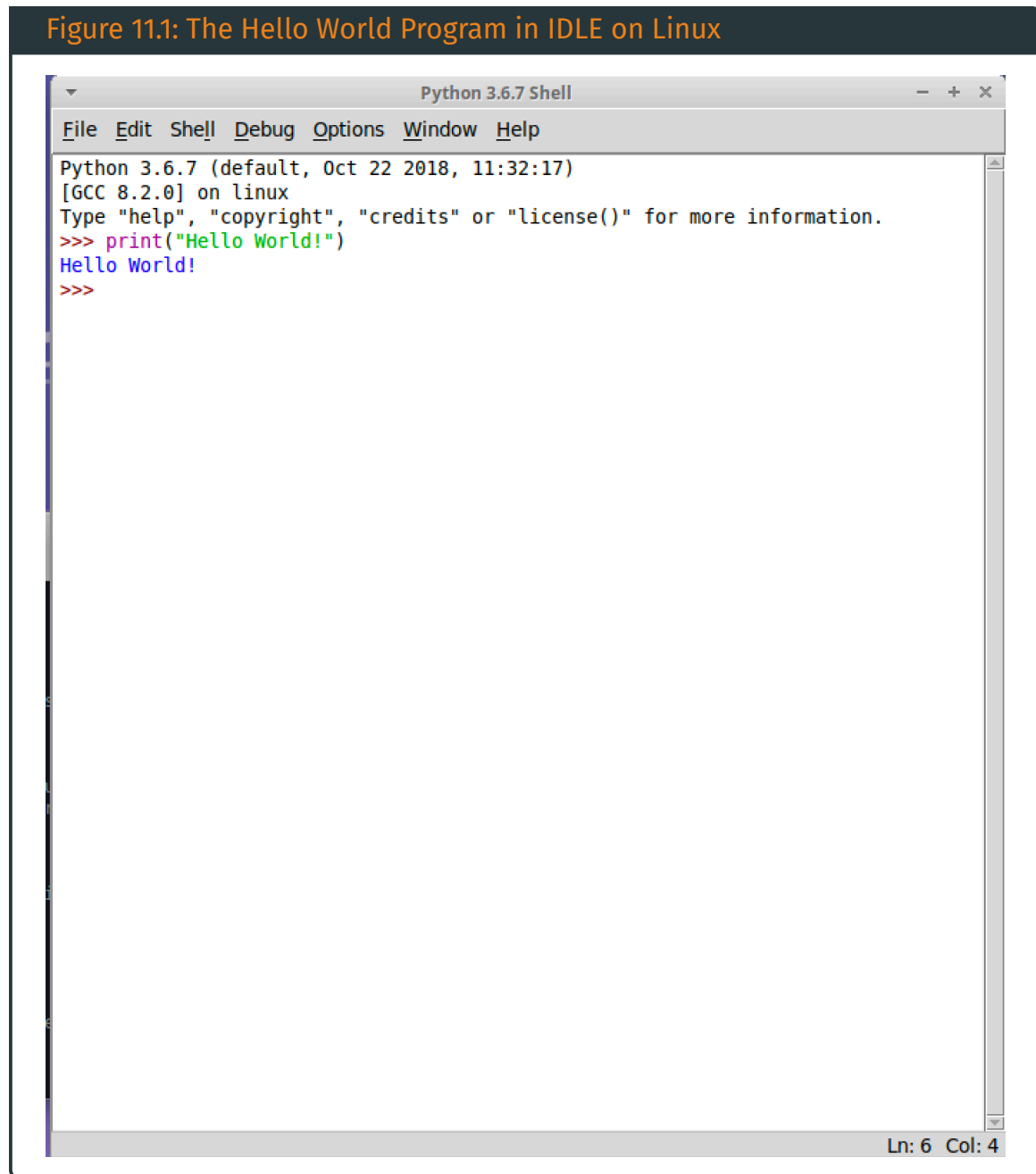
There is no shortage of Python programming environments that we can download and use. Some are *open source*<sup>1</sup>, and some are commercial. Some are very simple to get started with, but provide somewhat limited functionality for professionals; and others have lots of features but may be too complicated for the first-time programmer. In this course, we will be using one of the simple ones.

Appendix A, which starts on Page 97, details how to install **IDLE**, the development environment we will be using this semester. Once IDLE is running, we can type our program into the interactive Python interpreter, as seen in Figure 11.1 on Page 47.

---

<sup>1</sup>Stuff about FOSS

Figure 11.1: The Hello World Program in IDLE on Linux



```
Python 3.6.7 Shell
File Edit Shell Debug Options Window Help
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Hello World!")
Hello World!
>>>
```

Ln: 6 Col: 4

In class today, you will start up IDLE for the first time, and type some commands in.

## 11.3 Starting to Program in Python

As you've read in Chapter 1 of the textbook, most computer programs perform three basic kinds of instructions:

- Input

- Processing
- Output

Your instructor will guide you through some examples of each in class.

## Day 12

# Python Errors and zyLabs

Syntax, my lad. It has been restored to the highest place in the republic.  
– John Steinbeck

### Contents

12.1 Python Errors . . . . .	49
12.1.1 Syntax Errors . . . . .	50
12.1.2 Runtime Errors . . . . .	50
12.1.3 Semantic Errors . . . . .	51
12.2 zyLabs . . . . .	51

## 12.1 Python Errors

If you are an athlete, or a performer, or a writer, or a pursuer of any other creative enterprise, you already know that part of the exhilaration of performing is the danger that something won't go the way you expect. Programming a computer is no different. Finding an error in a computer program is a common occurrence. It has been nearly seventy years since a Major League Baseball player has successfully gotten a hit more than 40% of the time in one season. Baseball players don't look at outs as failures; they view them as opportunities to learn and improve. You must be prepared to do the same if you are going to be a successful programmer.

### 12.1.1 Syntax Errors

As you learned in Section 1.4 of zyBooks, and as we discussed way back on Day 2 (see Section 2.2.4 in these notes), there are a couple of different kinds of errors we programmers can make. The first kind of error is a **syntax error**. As we learned then, a syntax error comes from writing an instruction that does not conform to the language's rules.

Yesterday, you wrote code that looks like this:

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

As you now understand, that code is a *call* to the `print` function. A **function call** such as this requires the use of parentheses. If you tried to type this command without the parentheses, you would break the rules of Python's structure, and you would be informed of this immediately, as you see in Figure 12.1 on Page 52.

Notice that Python lets you know fairly clearly in red text that your error is a syntax error. Sometimes (but not always), Python will also include a helpful hint regarding how you can fix this error, as it has in this example.

As you learn about more of Python's features, be sure to pay attention to the syntax that is required to make use of these features. Like any programming language, Python can be very particular about how we type our code.

### 12.1.2 Runtime Errors

A **runtime error** is caused by a situation that the computer didn't expect. Since a runtime error occurs when a program runs, such an error can only occur in code that is free of syntax errors. Runtime errors typically cause the program to immediately terminate.

Consider this Python code that we looked at at the end of Day 11:

```
1 my_number = int(input('Type in a number: '))
2 print(my_number * 3)
```

That Python program is free of syntax errors, and will run just fine as long as the user interacts with it in an expected way. However, if you run the program and type Idaho instead of a number, the `int()` function won't be able to complete its job. Python will respond with this error message:

```
ValueError: invalid literal for int() with base 10: 'Idaho'
```

There are many kinds of runtime errors in Python, and you will not be required to know what they are. As we write more code, however, you will encounter more errors and of more types, and you will gain experience in finding and fixing them.

### 12.1.3 Semantic Errors

A **semantic error**, or *logic error*, is an error in a program's meaning. Like runtime errors, these errors only occur in a program that can run (although a good programmer can detect them ahead of time).

Consider this program:

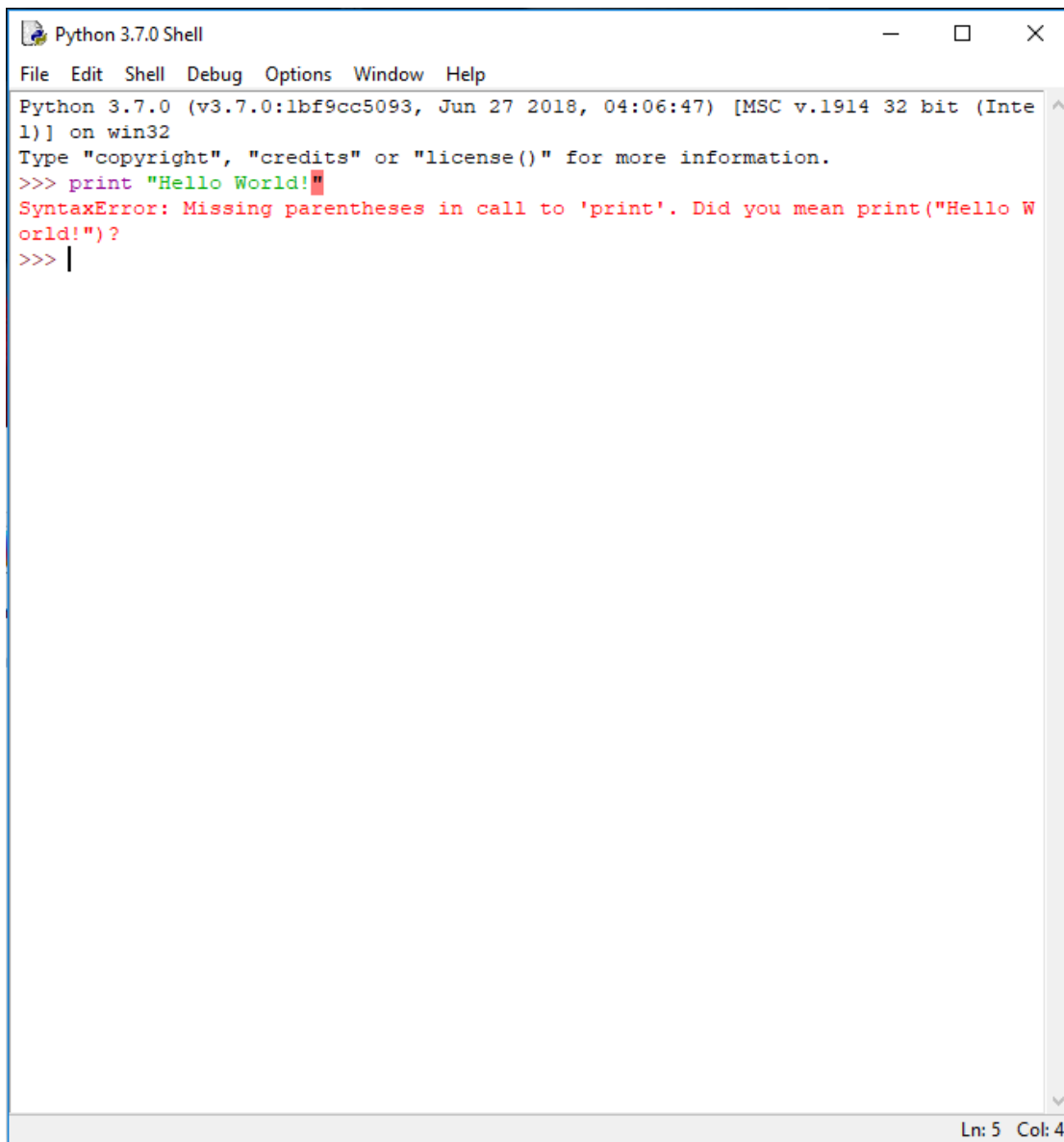
```
1 age = int(input('How old are you?'))
2 months = age / 12
3 print("That's", months, "months!")
```

That program will run to completion and provide nice-looking output, which happens to be wrong. The syntax is fine – if it weren't, the program would not have run. There were no runtime errors; the program provided its output and exited normally, with no red error messages. The problem here is that the programmer provided the wrong instructions. These kinds of errors are frequently referred to as *bugs*.

## 12.2 zyLabs

Your instructor will have assigned several interactive programming assignments, to be completed during today's class. These assignments are called **zyLabs**, and they're part of the zyBooks online textbook you signed up for. It is important that you read the instructions for each assignment carefully before starting, and follow them as precisely as you can, to maximize your grade – and to gain valuable hands-on experience programming in Python! Remember to have fun, and to ask your instructor for help as soon as you think you need to. There is no shame in asking for help, especially when trying to do something you have never done before.

Figure 12.1: Syntax Error: Calling a Function Without Parentheses



The image shows a screenshot of a Python 3.7.0 Shell window. The window has a title bar that says "Python 3.7.0 Shell" and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with options: File, Edit, Shell, Debug, Options, Window, and Help. The main text area contains the following text:

```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "Hello World!"
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("Hello World!")?
>>> |
```

The error message is displayed in red text. At the bottom right of the window, the status bar shows "Ln: 5 Col: 4".

## Day 13

# Variables and Expressions

Without memory, there is no culture. Without memory, there would be no civilization, no society, no future.  
– Elie Wiesel

### Contents

13.1 Variables . . . . .	53
13.1.1 Assignment . . . . .	53
13.1.2 A Variable Has One Value . . . . .	54
13.1.3 Identifiers . . . . .	54
13.2 Objects . . . . .	54
13.3 Two Kinds of Numbers . . . . .	54
13.4 Expressions . . . . .	55

## 13.1 Variables

As we started to explore on Day 11, a **variable** lets us store some information that we can use later. Specifically, when we create a variable, we ask the computer to use a part of its **memory**, or RAM, which we identify by name. Later uses of that name keep referring to the same place in memory, so that we can retrieve what we stored.

### 13.1.1 Assignment

We create a variable by assigning a value to it, like you did when you typed `name = 'Olivia'`. That equals sign in the middle has a special meaning to the



Python interpreter. It's called the **assignment operator**, and its presence in that statement makes it an **assignment statement**. We use assignment statements to create variables, and to change their values.

### 13.1.2 A Variable Has One Value

If, later in the program, we type `name = 'Henry'`, then the name Olivia is removed from memory, and the name Henry takes its place in the variable called name. If you need to store two pieces of information at the same time, you will have to make two variables, each with its own name.

### 13.1.3 Identifiers

Each variable must have a valid **identifier** – that's a fancy word for “name.” Chapter 2 of the textbook discusses what identifiers are legal according to Python syntax, and what makes other Python programmers think an identifier is good.

## 13.2 Objects

In Python, variables are considered to be **objects**, and each object has a value, a type, and an identity. The value is the data the object contains. The type of an object is the same as the type of its value. It's important to know that different data types sometimes disallow certain interactions – for instance, you can't add 5 to “Olivia”.

An object's identity allows us to tell whether two identifiers refer to the same object or not. This is not important now, but it will be later. What is important now is to see that an object's *identifier* is its name, and its *identity* is a reflection of how it is stored in memory.

We can use the Python function `type()` to get an object's type, and we can use the Python function `id()` to get an object's identity.

### 13.3 Two Kinds of Numbers

Programming languages differentiate between two different kinds of numbers. An **integer** is a number that has no fractional or decimal component. Programmers call numbers with decimal points **floating-point numbers**, because of how they're stored in memory. You will frequently hear programmers use the shortcuts `int` and `float` to refer to these data types, based on type names used in programming languages.

Just as there's an `int()` function to turn a string into an integer, there's a `float()` function to turn a string into a float.

In the next sections, you will learn to be careful when mixing numeric types in arithmetic expressions.

## 13.4 Expressions

You can build up all sorts of mathematical expressions using variable names, literal values (like 5), and operators. Notice that some operators are *unary*, like the negation operator in `-5`, while most operators are *binary*, like the subtraction operator in `x - 5`.

It is considered good programming practice to leave a space around mathematical operators, because this makes code easier to read. Consider:

```
1 # An expression with no whitespace
2 x=31*y-14*(z-2)/17
3
4 # The same expression with a space surrounding each operator
5 x = 31 * y - 14 * (z - 2) / 17
6
7 # Which one would you rather read?
```

Be sure to pay attention to the precedence rules when constructing an expression, because Python will always follow them when evaluating an expression.



# Day 14

## Modules

A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. That is the argument we have heard from our humanists and most of our computer scientists. Computer programs are good, they say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it.

– Marvin Minsky

### Contents

14.1	Where Your Files Go When You Save Them . . . . .	57
14.1.1	The Department's Computers . . . . .	57
14.1.2	Your Computer . . . . .	58
14.2	Three Kinds of Python Programs . . . . .	58

## 14.1 Where Your Files Go When You Save Them

### 14.1.1 The Department's Computers

When you log in to the Windows computers in the Department of Mathematics, Computer Science, and Information Technology's labs in B Cluster, these computers make a couple of interesting resources available to you. Of the most interest right now is a hard drive that is connected over the Department's network and mapped to drive J:\ (or "the J drive"). Because this is a network-mapped drive, you will always attach to the same J drive when you log in to any one of the computers in our computerized classrooms, or the Computer Learning Center in B 225, or even an instructor's computer in his or her faculty office.

For this reason, it is ***strongly suggested*** that you save all files you create on one of the Department's computers to the J drive – this way, there's less to remember.

It is important to know that any file you save to the C drive (the locally-mounted hard drive) on the Department's Windows computers is *very likely to not exist* when you go looking for it.

### 14.1.2 Your Computer

Do you use your own computer during class? Do you, or will you, use your own computer at home? If so, you may already have an organization system set up, and you don't have to think much about where your new Python programs are going to be saved. If not, then your computer is likely to suggest a suitable location.

In any case, make sure you know where your computer is storing files. If you aren't sure of the location, it will be incredibly difficult for someone else to help you find these files later.

## 14.2 Three Kinds of Python Programs

As you can see in Section 2.8 of zyBooks, there are three basic ways of using IDLE to create Python programs. So far, we have only been using the first, called *interactive mode*. Today, you will learn how to create **scripts** and **modules**. Be sure to read Sections 2.8 and 2.9 carefully before class, pay attention during lecture, and ask any questions you have to as soon as they come up. The work you do today will help you to understand these concepts, and create the tools you have to create when necessary.

## Day 15

# Strings and Numeric Types, and Writing a First Python Script

I'm not a great programmer; I'm just a good programmer with great habits.  
– Kent Beck

## Contents

15.1	Strings	59
15.2	Numeric Types	60
15.3	Writing a Python Script	60
15.4	The Header	60
15.4.1	The Shebang	60
15.4.2	The Docstring	61
15.4.3	The Rest of the Header	62

## 15.1 Strings

You will learn after reading Chapter 3 of the zyBooks textbook that a string is a sequence of characters, and that we can do some interesting processing with the string as a whole, and its individual components as well. After reading the chapter and completing the Participation Activities, you will be able to find the length of a string, compose a big string out of smaller strings, and determine which characters are in which places in a string.

## 15.2 Numeric Types

There are two types of real numbers in Python, `float` and `int`, which store different kinds of data and support different kinds of processing. Understanding the similarities and differences will help you not just with today's work, but with many of the tasks you will have to complete this semester.

## 15.3 Writing a Python Script

In today's class, you will be given a new kind of homework assignment. You will be tasked with writing a Python script and submitting it for grading. This assignment is not a zyLabs assignment, and you won't receive automated feedback like you do from zyLabs. Instead, you will write and test this script in IDLE, make sure for yourself that it works, and then submit it to your instructor for grading. As the semester goes on, you will have more and more assignments like this.

It is, of course, of critical importance that the script you submit performs the right task. However, we programmers demand more of ourselves than that. Program code must also be easily read by humans. As much as our source code conveys information to the computer, it must also convey information to us. Therefore, it's a good idea to agree upon a standard way for our code to be formatted, so that we can see this information easily.

## 15.4 The Header

While there is general consensus among the Python community, and the programming community generally, about what information should appear at the top of a source code file, there does not appear to be one standard regarding *how* this information should be presented. Presented here is the standard we will use, based upon commonly-used ideas and guidelines.

### 15.4.1 The Shebang

The idea of “scripting”, or “writing scripts”, has been around longer than Python has, and in fact has been around longer than Microsoft Windows has. Among the earliest kind of scripts are what are now referred to as “shell scripts”, because they are sets of instructions for the **command shell**, or the interactive program that waits for the computer's user to type in commands. In the earliest days of the Unix operating

system, before graphical user interfaces, all of the instructions to the computer had to be typed at the shell.<sup>1</sup>

Unix users started writing scripts as a way to automate the stuff they had to type repeatedly to accomplish tasks. These scripts need to start with a **shebang** to signal to Unix how to process the script. The shebang consists of a “sharp” symbol (think music notation – it’s the #) followed by a “bang” symbol, a programmer’s shorthand for the ! symbol. A sharp and a bang together came to be known as a shebang.<sup>2</sup>

A typical “shell script”, therefore, would have a first line something like this:

```
#!/usr/bin/sh
```

This tells Unix to use the program `sh`, located in the `/usr/bin` directory, to process the script. Since we’re working in Python, our shebang should look more like this:

```
#!/usr/bin/env python3
```

This tells your Unix system to use the `env` program to figure out where Python 3 is installed, and use it to process the script.

You might be wondering at this point, if the shebang is a Unix thing, what effect it has on a Python script running on Microsoft Windows. The developers of Python knew that people would want to run Python scripts on Windows, and so configured Python on Windows to safely ignore this line. Therefore, you should provide a shebang at the top of every Python script.

## 15.4.2 The Docstring

Every Python script and module – and certain program components that we will explore later – should contain a **docstring** that provides a description of the code that follows.

While it is possible to fit a docstring on a single line, this is not the accepted standard for a script or a module. We should include a multi-line docstring instead.

A docstring is really just a string, except that it starts and ends with *three quote symbols* instead of one, like this:

```
1 '''This is a docstring.
2
3 This is part of the same docstring.
4 '''
```

---

<sup>1</sup>The command shell in Unix is very powerful, and many programmers continue to choose to work with it rather than clicking things.

<sup>2</sup>Unix users have all sorts of fun names for things. See Eric Raymond’s Jargon File at <http://catb.org/jargon/html/>.



While the Python specification [PEP 257](#) provides the most rigorous description of what to include in a docstring at the top of a script, we shall summarize it here: The first line should contain a brief summary of the code, and then, following a blank line, the rest should consist of a more elaborate description. If the script or module you're creating is part of an assignment, include the due date in the description. The top of the Hello World script might, therefore, contain this after the shebang:

```
1 '''Display a friendly message.
2
3 Display a friendly "Hello World" message to the user,
4 confirming that the computer's Python installation is in good
5 working order.
6
7 This script is part of Homework #1, due 2019-09-09.
8 '''
```

### 15.4.3 The Rest of the Header

We shall define three special variables after the docstring (although in production code there are generally many more than these). Assign the variable `__author__` a string containing your name, assign `__section__` a string containing the section of the class you're enrolled in, and assign `__email__` a string containing your email address.

#### Code Example 15.1: Hello World With a Full Header

```
1 #! /usr/bin/env python3
2
3 '''Display a friendly message.
4
5 Display a friendly "Hello World" message to the user,
6 confirming that the computer's Python installation is in good
7 working order.
8
9 This script is part of Homework #1, due 2019-09-09.
10 '''
11
12 __author__ = 'J. Student'
13 __section__ = 'CSC 104 D1'
14 __email__ = 'N00100100@students.ncc.edu'
15
16 print("Hello world!")
```

## Day 16

# Tracing a Python Program

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.  
– Brian W. Kernighan

### 16.1 Tracing a Python Program

Successful programmers are adept at determining what a program will do just by reading the source code. This is true for several reasons, but one critical one is that it helps us find our mistakes. We tend to make assumptions about how a program will run, and when these assumptions are not borne out in the program's results, it is our job to find out why. In short, we must figure out the difference between *what we think* the program should do, and *what the program says* it will do.

Today's tasks will help you develop this skill. Specifically, you will learn how to determine what happens when the user types input into our program, how and when the data inside a variable changes, and how to describe what will appear on the screen when a program runs.

Take today's activities seriously – especially if you plan to take more Computer Science or Information Technology courses after this one – because you will rely on these skills over and over again.



## Day 17

# The `graphics.py` Module

If you can't make it good, at least make it look good.  
– Bill Gates

### Contents

17.1 A Very Brief Introduction to Classes and Objects . . . . .	65
17.1.1 Instantiating a Class With Additional Information . . . . .	66
17.2 Modules Revisited . . . . .	66
17.3 The <code>graphics.py</code> Module . . . . .	67

Today you will start learning how to make some fun, cool-looking Python programs. But first, you need to understand a few concepts.

## 17.1 A Very Brief Introduction to Classes and Objects

Python is an **object-oriented** programming language. This allows Python programmers to use our skills to describe groups of things – what object-oriented programmers call **classes** – based on how these things are similar. For instance, all students at Nassau are similar because they all have ID numbers and they all register for classes. In particular, when we define a class in our code, we specify the **behaviors** (like registering for classes) and the **properties** (like a student ID number) for a group of **objects**.

Once a programmer has defined a class, what usually follows is the process of creating one or more **instances** of that class. An instance of a class is just an object that has the properties and behaviors defined in that class. We create an instance by calling a special kind of function, which shares its name with the class.

Imagine that we have defined a class called `Pizza`. The code inside the `Pizza` class defines several properties of a pizza, like its size, and whether it's been cooked or not. The `Pizza` class also defines several behaviors for this pizza, like cooking it, or adding a topping.

Creating an instance of the `Pizza` class – an action programmers like to refer to as “instantiating a `Pizza`” – is quite easy:

```
1 my_first_pizza = Pizza()
```

Notice that the class is called `Pizza`, and so we call the `Pizza()` function to instantiate it.

### 17.1.1 Instantiating a Class With Additional Information

Not everyone eats plain pizza. If the `Pizza` class provides the functionality, you can specify the options you want when you instantiate it:

```
1 my_awesome_pizza = Pizza(pepperoni, sausage)
```

## 17.2 Modules Revisited

The world of Python code is broken up into **modules**. A module is just a file that has some Python code in it. Some modules contain the code for one class, and other modules contain the code for multiple classes.

Let's say our `Pizza` class exists in a module called `food.py`. We need to use an **import** statement to make the `Pizza` class available to us. There are two ways to import code. Consider this example:

```
1 import food
2
3 my_first_pizza = Pizza() # this isn't going to work
```

The reason that Line 3 isn't going to work is because we have to specify where the `Pizza()` function comes from. We can fix that example's syntax using the **dot operator**:

```
1 import food
2
3 my_first_pizza = Pizza.Pizza() # now this works
```

The dot operator is also sometimes called the *scope resolution operator*, because it helps the Python interpreter to resolve where it's supposed to find the `Pizza()` function. The dot operator always tells Python that the item after it is a feature of the item before it – in this case, `Pizza()` (the function) belongs to `Pizza` (the class).

This is fine, but it makes code take longer to type. Frequently, programmers use the other syntax for the **import** statement:

```
1 from food import Pizza
2
3 my_first_pizza = Pizza() # this works now
```

The **from ... import ...** syntax adds the class at the end of the statement to the local script's scope, so that we can call its functions without using the class name and the dot operator.

We will be using the **from ... import ...** syntax when we create graphical programs in this course.

## 17.3 The graphics.py Module

The `graphics.py` module contains Python classes written by Prof. John Zelle at Wartburg College in Iowa. You can always download a copy from <http://www.matcmp.ncc.edu/csc104/graphics.py>, but the original location is <http://mcsp.wartburg.edu/zelle/python/graphics.py>. To use this code, place a copy in the same location as the Python code you write. To make the classes available, just type:

```
from graphics import *
```

The main class from the `graphics` module is called `GraphWin`, and so the first thing you will have to do in any graphical program is *instantiate the GraphWin class* like this: `win = GraphWin('Title', 250, 250)`. Notice that the `GraphWin` function takes three *arguments* or *parameters*: a phrase which will appear in the title bar, the window's width (as an amount of pixels), and the window's height (also pixels).

Once an instance of the `GraphWin` class exists, we can add (or draw) other objects in it, like circles and other geometric shapes.

Code Example 17.1 is a simple program you can type in to make sure that you have installed the `graphics` module correctly. It instantiates a `GraphWin` object, a `Point` object, and a `Circle` object. If you installed the module successfully, then when you run this script you should see the circle in the middle of the window that gets created. Make sure this program works for you before coming to class on Day 17.

## Code Example 17.1: Testing the Graphics Module

```
1  #!/usr/bin/env python3
2
3  '''Test the graphics module.
4
5  Type this code in and run it to make sure you have
6  correctly installed the graphics module.
7  '''
8
9  __author__ = 'Your Name'
10 __section__ = 'Your Section'
11 __email__ = 'N00xxxxxx@students.ncc.edu'
12
13 from graphics import *
14
15 win = GraphWin('Testing the Graphics Module', 250, 250)
16
17 p = Point(125, 125)
18 c = Circle(p, 50)
19 c.setOutline('black')
20 c.setFill('red')
21 c.draw(win)
22
23 # Make the user have to click the mouse in
24 # the window before exiting
25 win.getMouse()
26 win.close()
```

## Day 18

# When Graphics Goes Wrong!

A computer is really just an instrument of expression, like a piano. It's how you play it that resolves what value it has.  
– Richard Taylor, the computer effects supervisor on the movie “Tron”

### Important Note: Get Your Work Finished

It is recommended that you have finished the colorful circles homework **before** starting today's class. It is further recommended that you refresh what you've learned about conditional statements, because you will be writing some in class today.

## 18.1 When Graphics Goes Right

The code in yesterday's Code Example makes a nice circle on the screen, but that's largely because we chose some appropriate values for sizes, locations, and colors. In a controlled environment, it's easy to make sure that a program will do what we expect.

## 18.2 What Can Go Wrong?

If, however, the point *p* wasn't placed at (125,125), but at (300,300), the program would not look the way we intend for it to look. What if those coordinates were not typed into the program – “hard coded,” as programmers would say – the way you saw them yesterday, but entered by a user? What if they were read in from a file? What



if we can't guarantee that the  $x$  and  $y$  values would fall between 0 and 249? Today, we explore these scenarios, and how we might handle them.

### 18.2.1 User Input

One way that programmers make programs more interesting is by allowing the program's user to interact with it. We do this by prompting for input, as you've seen before with the `input()` function. Remember that users are not perfect, and asking for their input – while necessary – can lead to unexpected situations. A successful programmer must be able to handle the unexpected.

## **Day 19**

### **Exam 2 Review**

Your instructor will have prepared some materials for you to review in class today.  
Make sure to ask questions!

## **Day 20**

### **Exam 2**

Good luck!



## **Part III**

# **Intermediate Python Programming**



# Day 21

## Python Conditionals

Truth can only be found in one place: the code.  
– Robert C. Martin

### Contents

21.1	Review: Conditional Statements . . . . .	75
21.2	Python Conditionals . . . . .	75
21.2.1	Write Real Statements . . . . .	76
21.2.2	Indentation . . . . .	76

### 21.1 Review: Conditional Statements

If you’ve already forgotten everything you learned last month about conditional statements, it is probably best for you to go back to Day 6 and re-read those sections.

### 21.2 Python Conditionals

The material from earlier this semester will have you well-prepared to handle writing if statements in Python. You already know how to identify the possible values and outcomes of a condition, and you largely already know how to write an if statement in Python. As you read through Chapter 3 in zyBooks, there is very little new information to be gleaned. Most importantly, pay attention to these items:

### 21.2.1 Write Real Statements

Where we previously wrote *pseudocode* like this:

```
if major = CS:
    "You're a CS major"
elif major = IT:
    "You're an IT major"
else:
    "Your major is something else"
```

The idea is there, but there are portions in that example that are not proper Python and won't compile.

#### Relational Operators

Recall that in Python, the `=` operator is used only for assignment. When comparing two values in Python, we must use the `==` operator.

#### String Literals

When determining whether a variable like `major` contains a value like "CS", that value must either be stored in a variable already, or surrounded by double quotes.

#### Displaying Output

Before the first exam, it was enough to just write "You're a CS major". Now, however, you've learned how to make a Python program display a string, so make sure to get that right.

### 21.2.2 Indentation

You will read about indentation in Chapter 3, but the important detail here is that you **must be consistent** in how you use indentation. Every statement at the same logical level must start in the same column of code. For this reason, most Python programmers recommend that you indent four spaces at a time.

Therefore, the previous example should look like this in Python:

```
1 if major == "CS":
2     print("You're a CS major")
3 elif major == "IT":
4     print("You're an IT major")
```

```
5 else:  
6     print("Your major is something else")
```





## Day 22

# Writing an Interactive Graphics Program

You don't learn to walk by following rules. You learn by doing, and by falling over.  
– Richard Branson

### Contents

22.1 Today's Activity . . . . .	79
---------------------------------	----

## 22.1 Today's Activity

Today's class will provide an opportunity for you to use what you've learned about the `graphics.py` package and conditional statements to write a program that recalls a zyLabs assignment from a few weeks ago.



# Day 23

## Python Functions

Algorithm: A word used by programmers when they do not want to explain what they did.  
– A programmer's t-shirt

### Contents

23.1 What Is a Function? . . . . .	81
23.1.1 What Do Functions Do? . . . . .	82
23.2 What Makes a Function Run? . . . . .	82
23.3 How Can We Tailor a Function to Our Needs? . . . . .	82

It is human nature to try to think about complex things in simple terms. When someone says “I made breakfast,” we know that such an activity takes several steps, but there is value in being able to convey that meaning with such a short sentence.

### 23.1 What Is a Function?

Chapter 5 of zyBooks defines a **function** as “...a named series of statements.” It is considered good programming practice to collect within a function a series of statements that performs a certain task, like calculating the area of a bedroom, or displaying a summary of the user’s inputs. It is also considered good programming practice to make sure that a function *only performs one task*. For instance, a function that calculates the area of a bedroom, stores the area in a variable, and displays the area to the screen should probably be rewritten as two functions – one that calculates and stores, and one that displays. This use of **modularity** – breaking a large solution down into parts – makes complex programs easier to design, because

you can assemble the small code segments you wish to use in an order that makes sense to you.

### 23.1.1 What Do Functions Do?

Some functions are designed to make interesting things appear on the screen. Some programmers use the term “output functions” to describe these. Other functions are used to make certain values get stored in certain variables. We will discuss both types in class.

## 23.2 What Makes a Function Run?

The code inside a function won't run until it is called. A **function call** is code we write in one part of a program to say we want code in a different part of a program to run. A function call must include the name of the function that needs to run, and possibly other information as well, depending on the function being called.

When the *caller* calls the function, the program's *flow of control* passes to the function. When the function is done running, flow of control passes back to the caller, and the code that appears after the function call runs next.

## 23.3 How Can We Tailor a Function to Our Needs?

A function that always calculates the area of the same bedroom would probably be of limited use to most programmers. If we could tell the function the lengths of the walls each time we call it, however, we could then reuse the same function many times to calculate the size of many different bedrooms. This idea, of sending different data to the same function to perform the same kind of calculation is referred to as the passing of **parameters**. For instance, we might call the function once and provide the parameters 10 and 8 (for a bedroom that measures ten feet by eight feet), and call it the next time with the parameters 9 and 12. Each call would *perform the same task* even though the answer the first time is 80 and the second time is 108.

## Day 24

# Tracing Python Functions

Based upon what we learned on Day [23](#), today we focus on following the flow of a program's control in and out of various functions.



## Day 25

# Validating User Input

### Contents

25.1 Introduction to Loops . . . . .	85
25.2 Iteration . . . . .	86
25.3 Syntax . . . . .	86

On Day 22 we wrote a program that asked for user input, and quit if the input was invalid. You are probably aware that real-world programs don't – or at least shouldn't – respond this way. It would be much more user-friendly if our program could give the user another chance instead of just exiting. Today we will learn a good technique for making this happen.

## 25.1 Introduction to Loops

In Section 2.3 of these notes, you learned that there are three fundamental kinds of computer statements: sequential statements, selection statements, and iteration statements. It is now time to learn about the last of these.

## 25.2 Iteration

One of the first goals of computer designers, even going back to Sir Charles Babbage,<sup>1</sup> was that they would make life simpler by doing the sort of tasks that we humans didn't want to do. In Chapter 4 of zyBooks, you will learn how Python makes it easy for us to tell the computer to automate certain tasks.

<sup>1</sup><https://www.computerhistory.org/babbage/charlesbabbage/>



## 25.3 Syntax

Take care to notice that the syntax of a **loop statement** is similar to that of a conditional statement: there's a keyword, a condition, a colon, and then a bunch of indented lines. See these examples:

### Code Example 25.1: If Statements and While Statements

```
1 x = 1
2
3 if x < 5:
4     print("Hello from the if statement!")
5     x += 1
6
7 print("More stuff happens here.")
8
9 while x < 5:
10    print("Hello from the while statement!")
11    x += 1
12
13 print("More stuff happens here.")
```

There's a big difference between the **if** statement and the **while** statement, however. The body of the **if** statement runs at most once, but the body of the **while** loop will keep repeating as long as the condition is true.

Before class, see if you can determine what the outcome of this program will be, and then type it in and run it to see if you were right.

## Day 26

# Graphics Programming Activity

A man who carries a cat by the tail learns something he can learn in no other way.  
– Mark Twain

Your instructor will have prepared a programming activity for you today in which you will be able to use most of the Python skills you have developed so far this semester. Be sure to ask questions as often as you have to, and remember that it's ok to experiment in code! If you're not sure if something will work, try it out! You can always erase it later.



## Day 27

# Finite Loops and Collections

Quote  
– Attribution

One of the most interesting things about computer programming is the range of human experiences that we can simulate. Modern programming languages provide us with great power in representing our world in meaningful ways. Today's lesson will explore how we can store and work with wide-ranging collections of data and information.



## Day 28

# Traffic Light Activity

Stopping at third adds no more to the score than striking out. It doesn't matter how well you start if you fail to finish.  
– *Billy Sunday*

The last programming activity of the semester will draw on everything you've learned! Do you have what it takes to create a working traffic light? Chances are that you do!



## Day 29

### Exam 3 Review

Your instructor will have prepared some materials for you to review in class today. Make sure to ask questions!

## Day 30

### Exam 3

Good luck! I hope this class met your expectations, and you feel ready for the next step of your journey!





# **Appendices**



# Appendix A

## IDLE

**IDLE** is a simple graphical development environment for the Python programming language. It is available for Windows, MacOS, and Linux.

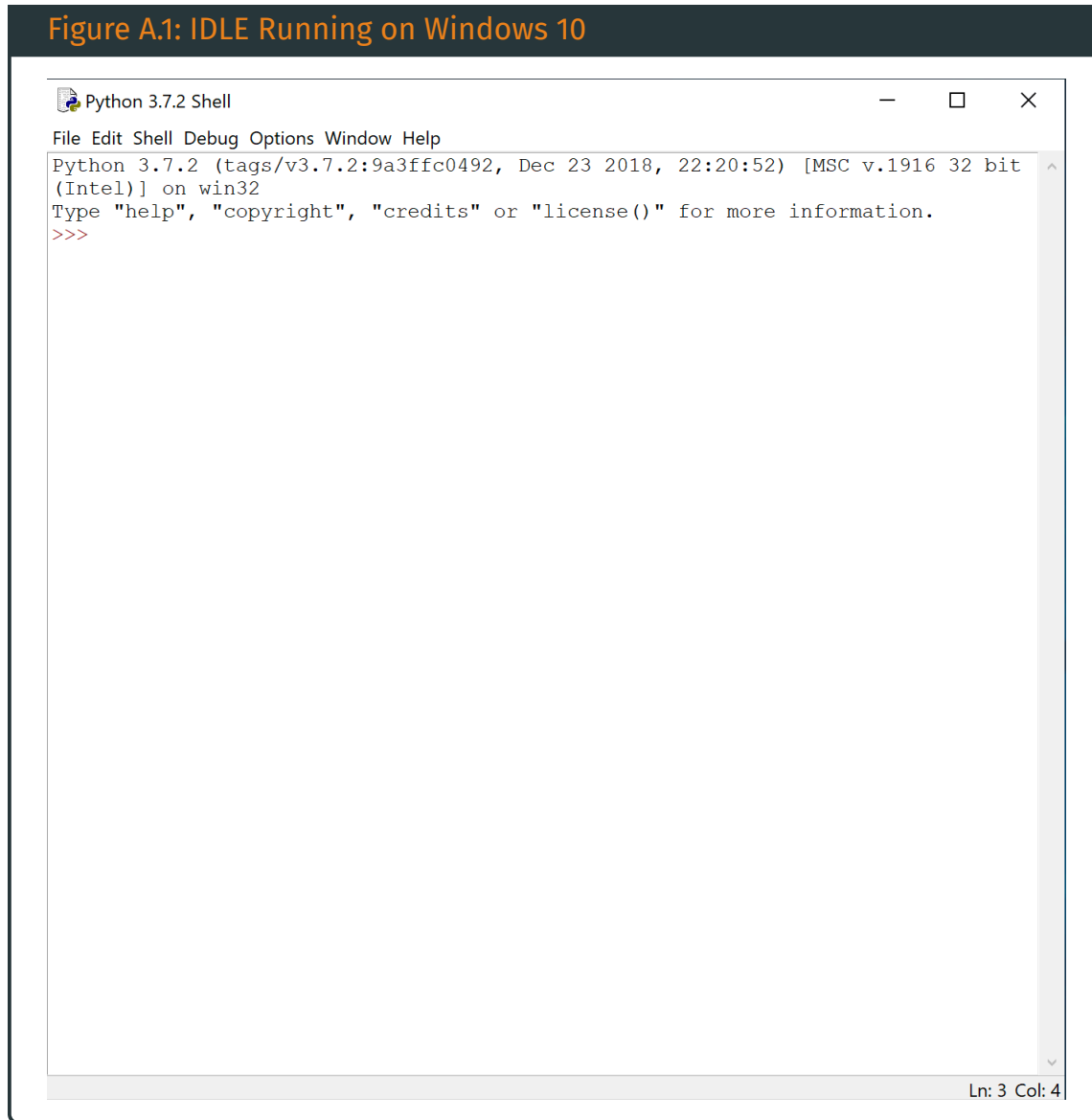
IDLE provides an *interactive* programming environment, as discussed in Section 1.2 of the textbook. It also provides a way for you to run programs that were prepared earlier and saved.

### A.1 Using IDLE

#### A.1.1 Windows

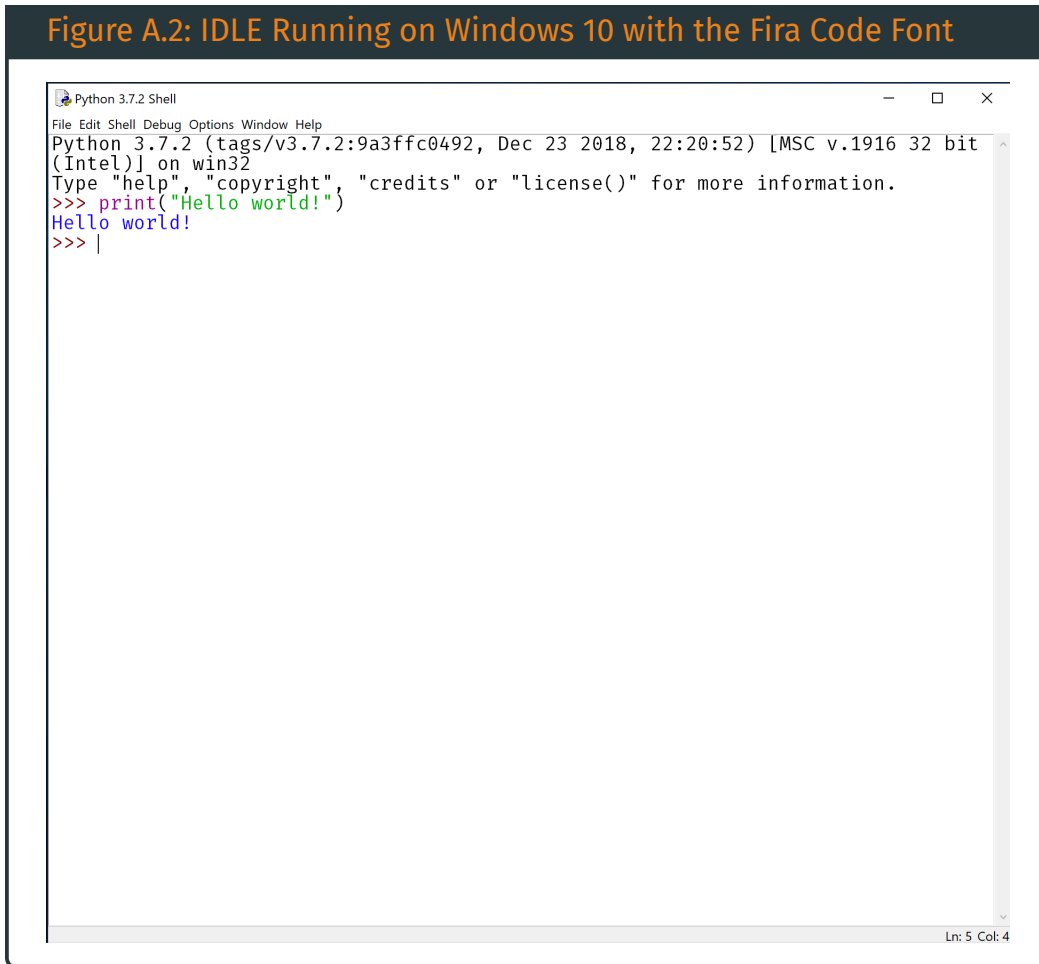
I downloaded Python from the official Python website, <https://www.python.org/downloads/>, and this download included IDLE. Make sure to download the most recent available version of Python. After running the installation program, I was able to run IDLE by pressing the Windows button in the lower left corner of the screen and then searching for IDLE. When you initially run the program, you should see a window like the one in Figure A.1 on Page 98.

Figure A.1: IDLE Running on Windows 10



Using the Option menu, I was able to change the font face to something nicer (I changed it to <https://www.fontsquirrel.com/fonts/fira-code>, the same font used for code examples in these notes) and the font size to something more readable, as you see in Figure A.2 on Page 99.

Figure A.2: IDLE Running on Windows 10 with the Fira Code Font



### A.1.2 Mac OS

There are a couple of ways to install Python and IDLE for Mac OS, but the easiest way is probably the same as in Windows: to download the installation program from <https://www.python.org/downloads/> and install.

### A.1.3 Linux

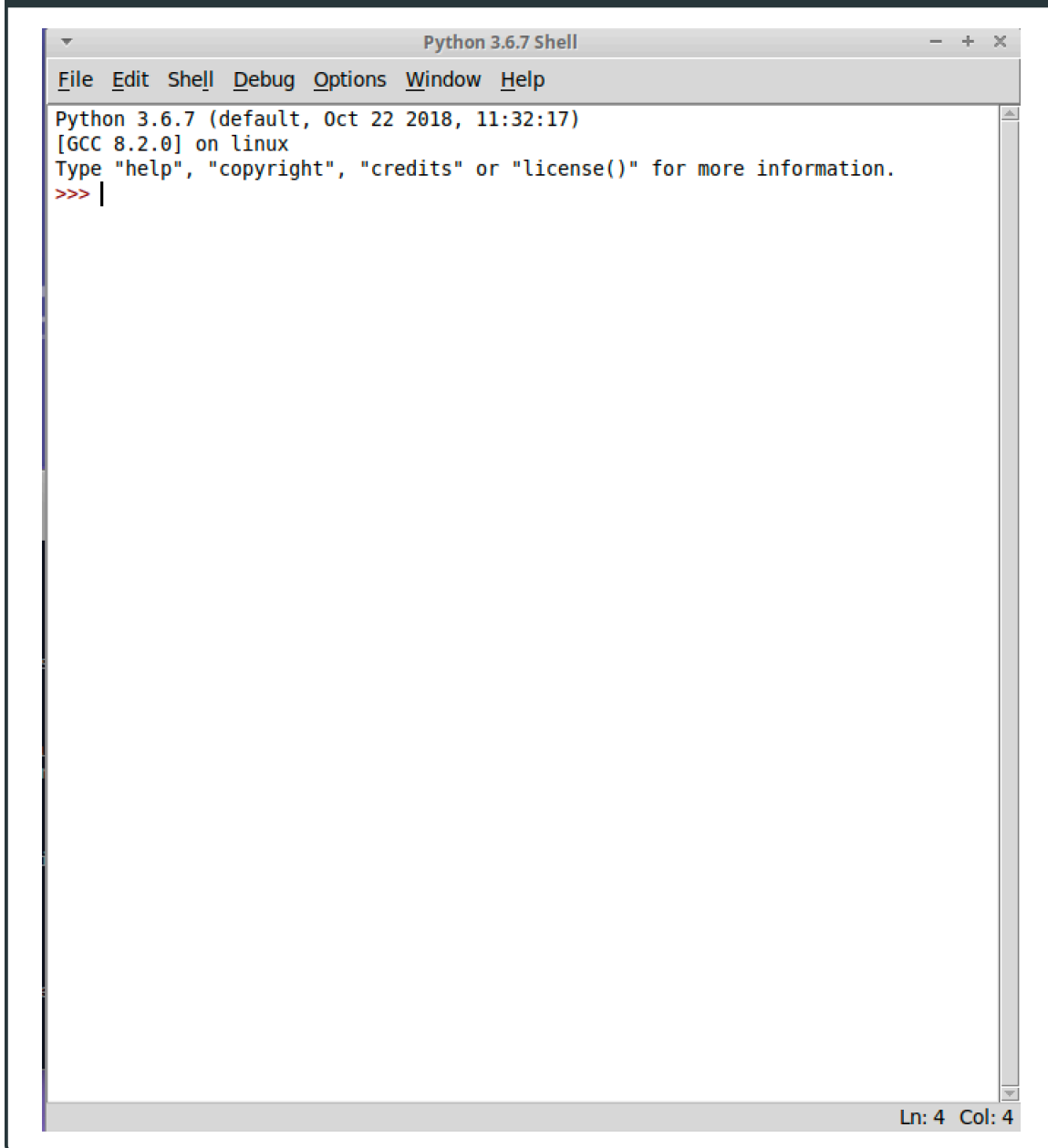
On my installation of Ubuntu Linux, I typed the following commands to install IDLE:

```
$ sudo apt update
$ sudo apt install idle
```

You could also use the graphical software-installation tool to install IDLE.

Once it's installed, just type `idle` at the command line prompt to open the program. You should see something like the window in Figure A.3 on Page 100.

Figure A.3: IDLE Running on Ubuntu Linux



It is important that you see the phrase “Python 3” at the top of the screen (in this screenshot, it says “Python 3.6.7”, and that’s good). If it says “Python 2” on the first line, then you are running an older version of IDLE – and therefore an older version of Python – and things won’t work the way you expect. If this is the case, try running `idle3` instead of just `idle`. If that doesn’t work, try to use `apt` to install `idle3`.

## Appendix B

### Python Keywords

**Keywords** are words that have a special meaning in Python. As discussed in XXXXX, we are not allowed to use keywords as identifiers.

This table was copied from this list at W3Schools: [https://www.w3schools.com/python/python\\_ref\\_keywords.asp](https://www.w3schools.com/python/python_ref_keywords.asp).

Notice that all of these keywords consist completely of lowercase letters except for False, True, and None.

Table B.1: The Keywords of the Python Language

Keyword	Meaning
and	A logical operator
as	To create an alias
assert	For debugging
break	To break out of a loop
class	To define a class
continue	To continue to the next iteration of a loop
def	To define a function
del	To delete an object
elif	Used in conditional statements, same as else if
else	Used in conditional statements
except	Used with exceptions, what to do when an exception occurs
False	Boolean value, result of comparison operations
finally	Used with exceptions, a block of code that will be executed no matter if there is an exception or not
for	To create a for loop
from	To import specific parts of a module
global	To declare a global variable

*Continued on next page*



Table B.1 – continued from previous page

Keyword	Meaning
if	To make a conditional statement
import	To import a module
in	To check if a value is present in a list, tuple, etc.
is	To test if two variables are equal
lambda	To create an anonymous function
None	Represents a null value
nonlocal	To declare a non-local variable
not	A logical operator
or	A logical operator
pass	A null statement, a statement that will do nothing
raise	To raise an exception
return	To exit a function and return a value
True	Boolean value, result of comparison operations
try	To make a try...except statement
while	To create a while loop
with	Used to simplify exception handling
yield	To end a function, returns a generator

# Index

- #!, [61](#)
- Algorithm, [10](#), [24](#)
- Binary numbers, [30](#)
- Bytecode, [13](#)
- Central Processing Unit, [11](#)
- Class, [65](#)
- Command shell, [60](#)
- Compiler, [13](#)
- Condition, [31](#)
- Conditional statement, *see* Statement
- CPU, *see* Central Processing Unit
- Decimal numbers, [29](#)
- Decision tree, [35](#)
- Docstring, [61](#)
- Dot operator, [66](#)
- Error
  - Runtime error, [50](#)
  - Semantic error, [12](#), [51](#)
  - Syntax error, [12](#), [50](#)
- Function, [81](#)
  - Call, [82](#)
  - Parameters, [82](#)
- Hard Drive, [11](#)
- Identifier, [54](#)
- IDLE, [46](#), [97](#)
- Import, [66](#)
- Input, [11](#)
- Instance, [65](#)
- Interpreter, [13](#)
- Keywords, [101](#)
- Matrix, [17](#)
- Memory, [11](#), [53](#)
- Modularity, [81](#)
- Module, [58](#), [66](#)
- Number
  - Floating point, [54](#)
  - Integer, [54](#)
- Object, [54](#)
- Object-Oriented Programming, [65](#)
- Operators
  - Boolean operators, [36](#)
  - Relational operators, [32](#)
- Output, [11](#)
- Program
  - Definition, [10](#)
  - Programming, [10](#)
- Reserved words, *see* Keywords
- Script, [58](#)
- Shebang, [61](#)
- Source code, [13](#)
- Statement
  - Conditional, [13](#), [32](#)
  - Complex, [36](#)
  - Nested, [35](#)
  - Iteration, *see* Loop
  - Loop, [14](#), [86](#)
  - Selection, [13](#)
  - Sequential, [13](#)
- Systematic List, [23](#)
- Variable, [53](#)

