

ZED SHAW'S HARD WAY SERIES



Early  
Release  
RAW & UNEDITED

Learn  
**PYTHON**  
*the* **HARD WAY**  
FIFTH EDITION



ZED A. SHAW

ZED SHAW'S HARD WAY SERIES



Early  
Release  
RAW & UNEDITED

Learn  
**PYTHON**  
*the* **HARD WAY**  
FIFTH EDITION



ZED A. SHAW

# **Learn Python the Hard Way: A Deceptively Simple Introduction to the Terrifyingly Beautiful World of Computers and Data Science, Fifth Edition**

**Zed A. Shaw**

## **A NOTE FOR EARLY RELEASE READERS**

With Early Release eBooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this title, please reach out to Pearson at [PearsonITAcademics@pearson.com](mailto:PearsonITAcademics@pearson.com)



# Contents

## *Preface*

### **Module 1: Getting Started in Python**

- Exercise 0: Gearing Up
- Exercise 1: A Good First Program
- Exercise 2: Comments and Pound Characters
- Exercise 3: Numbers and Math
- Exercise 4: Variables and Names
- Exercise 5: More Variables and Printing
- Exercise 6: Strings and Text
- Exercise 7: Combining Strings
- Exercise 8: Formatting Strings Manually
- Exercise 9: Multi-line Strings
- Exercise 10: Escape Codes in Strings
- Exercise 11: Asking People Questions
- Exercise 12: An Easier Way to Prompt
- Exercise 13: Parameters, Unpacking, Variables
- Exercise 14: Prompting and Passing
- Exercise 15: Reading Files
- Exercise 16: Reading and Writing Files
- Exercise 17: More Files

### **Module 2: The Basics of Programming**

- Exercise 18: Names, Variables, Code, Functions
- Exercise 19: Functions and Variables

Exercise 20: Functions and Files  
Exercise 21: Functions Can Return Something  
Exercise 22: Strings, Bytes, and Character Encodings  
Exercise 23: Introductory Lists  
Exercise 24: Introductory Dictionaries  
Exercise 25: Dictionaries and Functions  
Exercise 26: Dictionaries and Modules  
Exercise 27: The 5 Simple Rules to the Game of Code  
Exercise 28: Memorizing Logic  
Exercise 29: Boolean Practice  
Exercise 30: What If  
Exercise 31: Else and If  
Exercise 32: Making Decisions  
Exercise 33: Loops and Lists  
Exercise 34: While Loops  
Exercise 35: Branches and Functions  
Exercise 36: Designing and Debugging  
Exercise 37: Symbol Review

### **Module 3: Applying What You Know**

Exercise 38: Beyond Jupyter for Windows  
Exercise 39: Beyond Jupyter for macOS/Linux  
Exercise 40: Advanced Developer Tools  
Exercise 41: A Project Skeleton  
Exercise 42: Doing Things to Lists  
Exercise 43: Doing Things to Dictionaries  
Exercise 44: From Dictionaries to Objects  
Exercise 45: Basic Object-Oriented Programming  
Exercise 46: Inheritance and Advanced OOP

Exercise 47: Basic Object-Oriented Analysis and Design

Exercise 48: Inheritance versus Composition

Exercise 49: You Make a Game

Exercise 50: Automated Testing

## **Module 4: Python and Data Science**

Exercise 51: What Is Data Munging?

Exercise 52: Scraping Data from the Web

Exercise 53: Getting Data from APIs

Exercise 54: Data Conversion with Pandas

Exercise 55: How to Read Documentation (Featuring Pandas)

Exercise 56: Using Only Pandas

Exercise 57: The SQL Crash Course

Exercise 58: SQL Normalization

Exercise 59: SQL Relationships

Exercise 60: Advice from an Even Older Programmer

# Table of Contents

## *Preface*

## **Module 1: Getting Started in Python**

### Exercise 0: Gearing Up

General Instructions

Minimalist Start

Complete Instructions

Testing Your Setup

Learning the Command Line

Next Steps

### Exercise 1: A Good First Program

What You Should See

Study Drills

Common Student Questions

### Exercise 2: Comments and Pound Characters

What You Should See

Study Drills

Common Student Questions

### Exercise 3: Numbers and Math

What You Should See

Study Drills

Common Student Questions

### Exercise 4: Variables and Names

What You Should See

Study Drills

Common Student Questions



## Exercise 5: More Variables and Printing

What You Should See

Study Drills

Common Student Questions

## Exercise 6: Strings and Text

What You Should See

Study Drills

Break It

Common Student Questions

## Exercise 7: Combining Strings

What You Should See

Study Drills

Break It

Common Student Questions

## Exercise 8: Formatting Strings Manually

What You Should See

Study Drills

Common Student Questions

## Exercise 9: Multi-line Strings

What You Should See

Study Drills

Common Student Questions

## Exercise 10: Escape Codes in Strings

What You Should See

Escape Sequences

Study Drills

Common Student Questions

## Exercise 11: Asking People Questions

What You Should See

Study Drills

Common Student Questions

Exercise 12: An Easier Way to Prompt

What You Should See

Study Drills

Common Student Questions

Exercise 13: Parameters, Unpacking, Variables

Code Description

Hold Up! Features Have Another Name

What You Should See

Study Drills

Common Student Questions

Exercise 14: Prompting and Passing

What You Should See

Study Drills

Common Student Questions

Exercise 15: Reading Files

What You Should See

Study Drills

Common Student Questions

Exercise 16: Reading and Writing Files

What You Should See

Study Drills

Common Student Questions

Exercise 17: More Files

What You Should See

Study Drills

Common Student Questions

## **Module 2: The Basics of Programming**

Exercise 18: Names, Variables, Code, Functions

- Exercise Code
- What You Should See
- Study Drills
- Common Student Questions

Exercise 19: Functions and Variables

- What You Should See
- Study Drills
- Common Student Questions

Exercise 20: Functions and Files

- What You Should See
- Study Drills
- Common Student Questions

Exercise 21: Functions Can Return Something

- What You Should See
- Study Drills
- Common Student Questions

Exercise 22: Strings, Bytes, and Character Encodings

- Initial Research
- Switches, Conventions, and Encodings
- Dissecting the Output
- Dissecting the Code
- Encodings Deep Dive
- Breaking It

Exercise 23: Introductory Lists

- Accessing Elements of a List
- Practicing Lists
- The Code
- The Challenge
- Final Challenge

Exercise 24: Introductory Dictionaries

- Key/Value Structures
- Combining Lists with Data Objects
- The Code
- What You Should See
- The Challenge
- Final Challenge

#### Exercise 25: Dictionaries and Functions

- Step 1: Function Names Are Variables
- Step 2: Dictionaries with Variables
- Step 3: Dictionaries with Functions
- Step 4: Deciphering the Last Line
- Study Drill

#### Exercise 26: Dictionaries and Modules

- Step 1: Review of `import`
- Step 2: Find the `__dict__`
- Step 3: Change the `__dict__`
- Study Drill: Find the “Dunders”

#### Exercise 27: The 5 Simple Rules to the Game of Code

- Rule 1: Everything Is a Sequence of Instructions
- Rule 2: Jumps Make the Sequence Non-Linear
- Rule 3: Tests Control Jumps
- Rule 4: Storage Controls Tests
- Rule 5: Input/Output Controls Storage
- Putting It All Together

#### Exercise 28: Memorizing Logic

- The Truth Terms
- The Truth Tables
- Common Student Questions

#### Exercise 29: Boolean Practice

- What You Should See

Study Drills

Common Student Questions

Exercise 30: What If

What You Should See

Study Drill

Common Student Questions

Exercise 31: Else and If

What You Should See

Study Drills

Common Student Questions

Exercise 32: Making Decisions

What You Should See

dis() It

Study Drills

Common Student Questions

Exercise 33: Loops and Lists

What You Should See

Study Drills

Common Student Questions

Exercise 34: While Loops

What You Should See

Study Drills

Common Student Questions

Exercise 35: Branches and Functions

What You Should See

Study Drills

Common Student Questions

Exercise 36: Designing and Debugging

From Idea to Working Code

Rules for If-Statements

Rules for Loops  
Tips for Debugging  
Homework

#### Exercise 37: Symbol Review

Keywords  
Data Types  
String Escape Sequences  
Old-Style String Formats  
Operators  
Reading Code  
Study Drills  
Common Student Questions

### **Module 3: Applying What You Know**

Exercise 38: Beyond Jupyter for Windows  
Exercise 39: Beyond Jupyter for macOS/Linux  
Exercise 40: Advanced Developer Tools  
Exercise 41: A Project Skeleton  
Exercise 42: Doing Things to Lists  
Exercise 43: Doing Things to Dictionaries  
Exercise 44: From Dictionaries to Objects  
Exercise 45: Basic Object-Oriented Programming  
Exercise 46: Inheritance and Advanced OOP  
Exercise 47: Basic Object-Oriented Analysis and Design  
Exercise 48: Inheritance versus Composition  
Exercise 49: You Make a Game  
Exercise 50: Automated Testing

### **Module 4: Python and Data Science**

Exercise 51: What Is Data Munging?



Exercise 52: Scraping Data from the Web

Exercise 53: Getting Data from APIs

Exercise 54: Data Conversion with Pandas

Exercise 55: How to Read Documentation (Featuring Pandas)

Exercise 56: Using Only Pandas

Exercise 57: The SQL Crash Course

Exercise 58: SQL Normalization

Exercise 59: SQL Relationships

Exercise 60: Advice from an Even Older Programmer

# **Preface**

**This content is currently in development.**

# **Module 1: Getting Started in Python**

## Exercise 0. Gearing Up

This exercise has no code. It is simply the exercise you complete to get your computer to run Python. You should follow these instructions as exactly as possible. If you have problems following the written instructions, then watch the included videos for your platform.

### General Instructions

Your general task is to get a “programming environment” with tools you can use to write code. Nearly every programmer has their own specialized environment, but at first you’ll want something simple that can get you through this course. After the course you’ll know enough about programming to then waste the rest of your life trying every tool you can imagine. It’s a lot of fun.

What you’ll need is the following:

- [Jupyter](#), which will be used in the first part of the book to get you started easily. Jupyter is a programming and data analysis environment that uses many languages, but we’ll use Python.
- Python. The version of Python you install *mostly* doesn’t matter so long as it’s older than version 3.6. Versions of Python (and other software) use numbers to indicate their age, and the position of the numbers determines how much changed between versions. The general rule is the first number means “major change,” the second number means “minor changes,” and a third number means only bug or security fixes. That means if you have version 3.8 and version 3.10, then there are no major changes, but there are minor changes. If you have versions 3.10.1 and 3.10.2, then there are only bug fixes.
- A *basic* programmer’s editor. Programmers use very complicated text editors, but you should start with something simple that still works as a

programmer's editor.

- A Terminal emulator. This is a text-based command interface to your computer. If you've ever seen a movie with hacking and programming in it, you've seen people furiously typing green text into a black screen so they can take down an entire alien race with their "unix exe 32 pipe attack." At first you won't need this, but later you'll "graduate" to using the Terminal as it's incredibly powerful and not too hard to learn.

You should have most of the other things you'll need on your computer already, so let's install each of these requirements for your operating system (OS).

## Minimalist Start

The instructions in this exercise are designed to install most of the things you need for the rest of the course, but if you want to get going quickly with the least amount of work, then install:

1. [Anaconda](#) to get your Python.
2. [Jupyter](#) to write and run some code.
  - a. On Windows the best way to run Jupyter is to hit the Windows key (Start menu) and type `jupyter -lab`. This will start it in a way that makes sense.
  - b. On Linux it should be the same command in your Terminal.
  - c. On macOS you can either type that command in the Terminal or start the app like normal.

This will give you enough to get started, but eventually you'll hit exercises that need the Terminal and Python from the "command line." Come back to this exercise when you reach that point in the course.

## Complete Instructions

Eventually you'll need to install more software to complete the course. The problem with installation instructions in books is they become outdated quickly. To solve this problem, I have a web page you need to visit with all of the instructions for your OS with videos showing you the installations. These instructions are updated whenever things change, and the web page includes any errata needed for your book.

To view these instructions, visit the following link:

- <https://learncodethehardway.com/setup/python/>

If you're not able to visit this link for some reason, then here's what you'll have to install:

1. [Anaconda](#) to get your Python.
2. [Jupyter](#) to write and run some code.
3. [Geany](#) for editing text later.
4. On Windows use the full install of [Cmder](#) as your shell.
5. On macOS you have Terminal, and Linux has whatever you want.

## Testing Your Setup

Once you have everything installed, go through these steps to confirm that everything is working:

1. Start your Terminal and type this command exactly, spaces and all:  
`mkdir lpthw.`
2. Once that works, you have a directory `lpthw` where you can place your work.
3. Go into that directory with the command `cd lpthw`. This command “moves” your Terminal into that directory so your work is saved there.
4. A “directory” is also called a “folder” on Windows and macOS. You can make the connection between the “directory” in your Terminal



and the “folder” you normally see by typing `start .` on Windows or `open .` on macOS. This opens the current directory into a graphical folder window you’re used to normally seeing. If you’re ever lost, type that.

5. The `start` command (open on macOS) works like double-clicking that thing with your mouse. If you are in the Terminal and want to “open” something, just use this command. Let’s say there’s a text file named `test.txt` and you want to open it in your editor. Type `start test.txt` on Windows or `open test.txt` on macOS.
6. Now that you can open your Terminal and open things while in your Terminal, you’ll want to start your editor. This is Geany if you’ve been following instructions. Start it and create a file named `test.txt` and then save it in the `lpthw` directory you made. If you can’t find it, remember you can open it from the Terminal with `start` (open on macOS) and then use that folder window to find it.
7. Once you’ve saved the file in the `lpthw` directory, you should be able to type `ls test.txt` in your Terminal to see that it’s there. If you get an error, then either you’re not in the `lpthw` directory and need to type `cd ~/lpthw` or you saved it in the wrong place. Try again until you can do this.
8. Finally, in the Terminal, type `jupyter -lab` to start Jupyter and make sure it works. It should open your web browser, and then you’ll see the Jupyter app inside your browser. It’s kind of like a little website on your personal computer.

Think of these tasks as a kind of puzzle to solve. If you get stuck, you should watch the video for your OS where I show you how to do everything. I think the best way to learn this is to first attempt it yourself; then when you get stuck, watch the video to see how I did it.

## Learning the Command Line

You don't need to do this right now, but if you're struggling with the previous tasks, you might need to go through the [Command Line](#) Crash Course to learn the basics of the Terminal (also called the "command line"). You won't need these skills for a while, but the command line is a very good introduction to controlling your computer with words. It will also help you with many other tasks in programming later, so learning it now can only help.

## Next Steps

Once you have everything working, you can continue with the rest of the course. If you ever run into trouble, you can email me at [help@learncodethehardway.com](mailto:help@learncodethehardway.com), and I'll help you. When you email me for help, take the time to describe your problem in as much detail as possible, and include screenshots.

# Exercise 1. A Good First Program

---

## Warning!

If you skipped [Exercise 0](#), then you are not doing this book right. Are you trying to use IDLE or an IDE? I said not to use one in [Exercise 0](#), so you should not use one. If you skipped [Exercise 0](#), please go back to it and read it.

---


You should have spent a good amount of time in [Exercise 0](#) learning how to install Jupyter, run Jupyter, run the Terminal, and work with both of them. If you haven't done that, then do not proceed. You will not have a good time. This is the only time I'll start an exercise with a warning that you should not skip or get ahead of yourself.

Type the following text into a Jupyter cell:

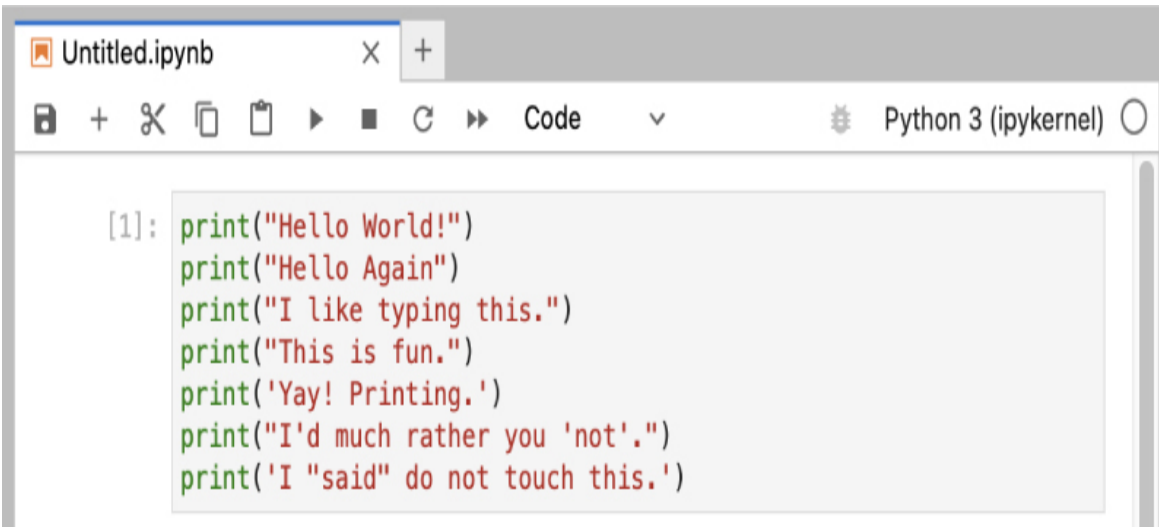
Listing 1.1: ex1.py

---

```
1  print("Hello World!")
2  print("Hello Again")
3  print("I like typing this.")
4  print("This is fun.")
5  print('Yay! Printing.')
6  print("I'd much rather you 'not'.")
7  print('I "said" do not touch this.')
```



Your Jupyter cell should look something like this:



```
[1]: print("Hello World!")
print("Hello Again")
print("I like typing this.")
print("This is fun.")
print('Yay! Printing.')
print("I'd much rather you 'not'.")
print('I "said" do not touch this.')
```

Don't worry if your Jupyter window doesn't look exactly the same; it should be close though. You may have a slightly different window header, maybe slightly different colors, and the left side of your Jupyter window won't be the same, but will instead show the directory you used for saving your files. All of those differences are fine.

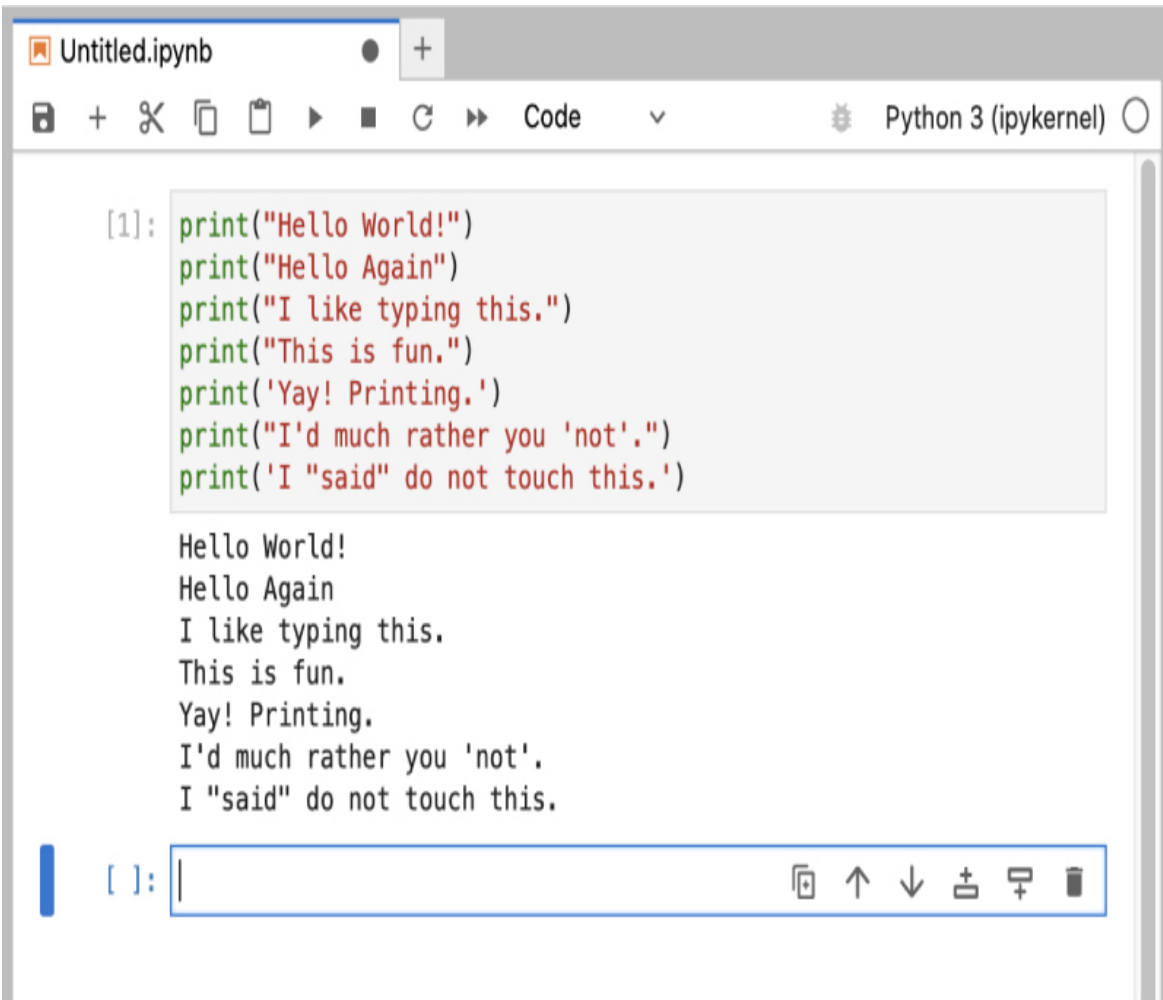
When you create this cell, keep in mind these points:

1. I did not type the line numbers on the left. Those are printed in the book so I can talk about specific lines by saying, "See line 5..." You do not type line numbers into Python scripts.
2. I have the `print` at the beginning of the line, and it looks exactly the same as what I have in the cell. Exactly means exactly, not kind of sort of the same. Every single character has to match for it to work. Color doesn't matter, only the characters you type.

Once it is *exactly* the same, you can hit SHIFT-ENTER to run the code. If you did it right, then you should see the same output as I in the *What You Should See* section of this exercise. If not, you have done something wrong. No, the computer is not wrong.

## What You Should See

The Jupyter output will look like this after you hold SHIFT and hit ENTER (which I'll write as SHIFT-ENTER):



```
[1]: print("Hello World!")
      print("Hello Again")
      print("I like typing this.")
      print("This is fun.")
      print('Yay! Printing.')
      print("I'd much rather you 'not'.")
      print('I "said" do not touch this.')

Hello World!
Hello Again
I like typing this.
This is fun.
Yay! Printing.
I'd much rather you 'not'.
I "said" do not touch this.
```

You may see different window appearance and layout, but the important part is that you type the command and see the output is the same as mine.

If you have an error, it will look like this:

```
1  Cell In[1], line 3
2      print("I like typing this.
3          ^
4  SyntaxError: unterminated string literal (detected at line 1)
```

It's important that you can read these error messages because you will be making many of these mistakes. Even I make many of these mistakes. Let's look at this line by line.

1. We ran our command in the Jupyter cell with SHIFT-ENTER.
2. Python tells us that the cell has an error on line 3.

3. It prints this line of code for us to see it.
4. Then it puts a ^ (caret) character to point at where the problem is. Notice the missing " (double-quote) character the end though?
5. Finally, it prints out a "SyntaxError" and tells us something about what might be the error. Usually these errors are very cryptic, but if you copy that text into a search engine, you will find someone else who's had that error, and you can probably figure out how to fix it.

## Study Drills

The Study Drills contain things you should *try* to do. If you can't, skip it and come back later.

For this exercise, try these things:

1. Make your script print another line.
2. Make your script print only one of the lines.
3. Put a # (octothorpe) character at the beginning of a line. What did it do? Try to find out what this character does.

From now on, I won't explain how each exercise works unless an exercise is different.

---

### Info

An "octothorpe" is also called a "pound," "hash," "mesh," or any number of names. Pick the one that makes you chill out.

---

## Common Student Questions

These are *actual* questions that real students have asked when doing this exercise:



**Can I use IDLE?** No, for now just use Jupyter and later we'll use a regular text editor for extra superpowers.

**Editing the code in Jupyter is annoying. Can I use a text editor?**

Totally, you can also create a python file in Jupyter and get a “good enough” editor. In the left panel where you see all your files, click the + (plus) icon on the top left. That will bring you to the first screen you saw when you started Jupyter. Down at the bottom under `$_` other you'll see a button for Python File with the Python logo. Click that and you'll get an editor to work on your file.

**My code doesn't run; I just get the prompt back with no output.**

You most likely took the code in my cell literally and thought that `print("Hello World!")` meant to type only "Hello World!" into the cell, without the `print`. Your cell has to be *exactly* like mine.

## Exercise 2. Comments and Pound Characters

Comments are very important in your programs. They are used to tell you what something does in English, and they are used to disable parts of your program if you need to remove them temporarily. Here's how you use comments in Python:

Listing 2.1: ex2.py

```
1  # A comment, this is so you can read your program later.
2  # Anything after the # is ignored by python.
3
4  print("I could have code like this.") # and the comment
5
6  # You can also use a comment to "disable" or comment out
7  # print(*("This won't run."))*
8
9  print("This will run.")
```

From now on, I'm going to write code like this. It is important for you to understand that everything does not have to be literal. If my Jupyter looks a little different from yours or if I'm using a text editor, the results will be the same. Focus more on the textual output and less on the visual display such as fonts and colors.

### What You Should See

```
1  I could have code like this.
2  This will run.
```

Again, I'm not going to show you screenshots of all the Terminals possible. You should understand that the preceding is not a literal translation of what

your output should look like visually, but the text is what you focus on.

## Study Drills

1. Find out if you were right about what the # character does and make sure you know what it's called (octothorpe or pound character).
2. Take your code and review each line going backward. Start at the last line, and check each word in reverse against what you should have typed.
3. Did you find more mistakes? Fix them.
4. Read what you typed out loud, including saying each character by its name. Did you find more mistakes? Fix them.

## Common Student Questions

**Are you sure # is called the pound character?** I call it the octothorpe because that is the only name that no one country uses and that works in every country. Every country thinks its name for this one character is both the most important way to do it and the only way it's done. To me this is simply arrogance and, really, y'all should just chill out and focus on more important things like learning to code.

**Why does the # in `print("Hi # there.")` not get ignored?** The # in that code is inside a string, so it will be put into the string until the ending " character is hit. Pound characters in strings are just considered characters, not comments.

**How do I comment out multiple lines?** Put a # in front of each one.

**I can't figure out how to type a # character on my country's keyboard. How do I do that?** Some countries use the ALT key and combinations of other keys to print characters foreign to their language. You'll have to search online to see how to type it.

**Why do I have to read code backward?** It's a trick to make your brain not attach meaning to each part of the code, and doing that makes you process each piece exactly. This catches errors and is a handy error-checking technique.

## Exercise 3. Numbers and Math

Every programming language has some kind of way of doing numbers and math. Do not worry: programmers frequently lie about being math geniuses when they really aren't. If they were math geniuses, they would be doing math, not writing buggy web frameworks so they can drive race cars.

This exercise has lots of math symbols. Let's name them right away so you know what they are called. As you type this one in, say the name. When saying them feels boring, you can stop saying them. Here are the names:

- + plus
- - minus
- / slash
- \* asterisk
- % percent
- < less-than
- > greater-than
- <= less-than-equal
- >= greater-than-equal

Notice how the operations are missing? After you type in the code for this exercise, go back and figure out what each of these does and complete the table. For example, + does addition.

Listing 3.1: ex3.py

---

```
1  print("I will now count my chickens:")
2
3  print("Hens", 25 + 30 / 6)
4  print("Roosters", 100 - 25 * 3 % 4)
```

```
5
6  print("Now I will count the eggs:")
7
8  print(3 + 2 + 1 - 5 + 4 % 2 - 1 / 4 + 6)
9
10 print("Is it true that 3 + 2 < 5 - 7?")
11
12 print(3 + 2 < 5 - 7)
13
14 print("What is 3 + 2?", 3 + 2)
15 print("What is 5 - 7?", 5 - 7)
16
17 print("Oh, that's why it's False.")
18
19 print("How about some more.")
20
21 print("Is it greater?", 5 > -2)
22 print("Is it greater or equal?", 5 >= -2)
23 print("Is it less or equal?", 5 <= -2)
```

Make sure you type this exactly before you run it. Compare each line of your file to my file.

## What You Should See

```
1  I will now count my chickens:
2  Hens 30.0
3  Roosters 97
4  Now I will count the eggs:
5  6.75
6  Is it true that 3 + 2 < 5 - 7?
7  False
8  What is 3 + 2? 5
9  What is 5 - 7? -2
10 Oh, that's why it's False.
11 How about some more.
12 Is it greater? True
```



```
13  Is it greater or equal? True
14  Is it less or equal? False
```

## Study Drills

1. Above each line, use the `#` to write a comment to yourself explaining what the line does.
2. You can type most math directly into a Jupyter cell and get results. Try using it to do some basic calculations like `1+2` and hit `SHIFT-ENTER`.
3. Find something you need to calculate and write a new `.py` file that does it.
4. Rewrite this exercise to use floating point numbers so it's more accurate. `20.0` is floating point.

## Common Student Questions

**Why is the `%` character a “modulus” and not a “percent”?** Mostly that's just how the designers chose to use that symbol. In normal writing you are correct to read it as a “percent.” In programming this calculation is typically done with simple division and the `/` operator. The `%` modulus is a different operation that just happens to use the `%` symbol.

**How does `%` work?** Another way to say it is, “X divided by Y with J remaining.” For example, “100 divided by 16 with 4 remaining.” The result of `%` is the J part, or the remaining part.

**What is the order of operations?** In the United States we use an acronym called PEMDAS which stands for Parentheses Exponents Multiplication Division Addition Subtraction. That's the order Python follows as well. The mistake people make with PEMDAS is to think this is a strict order, as in “Do P, then E, then M, then D, then A, then S.” The actual order is you do the multiplication *and*

division (M&D) in one step, from left to right, and *then* you do the addition and subtraction in one step from left to right. So, you could rewrite PEMDAS as PE(M&D)(A&S).

## Exercise 4. Variables and Names

Now you can print things with `print`, and you can do math. The next step is to learn about variables. In programming, a variable is nothing more than a name for something, similar to how my name “Zed” is a name for “the human who wrote this book.” Programmers use these variable names to make their code read more like English and because they have lousy memories. If they didn’t use good names for things in their software, they’d get lost when they tried to read their code again.

If you get stuck with this exercise, remember the tricks you have been taught so far for finding differences and focusing on details:

1. Write a comment above each line explaining to yourself what it does in English.
2. Read your Python code backward.
3. Read your Python code out loud, saying even the characters.

Listing 4.1: ex4.py

```
1  cars = 100
2  space_in_a_car = 4.0
3  drivers = 30
4  passengers = 90
5  cars_not_driven = cars - drivers
6  cars_driven = drivers
7  carpool_capacity = cars_driven * space_in_a_car
8  average_passengers_per_car = passengers / cars_driven
9
10
11  print("There are", cars, "cars available.")
12  print("There are only", drivers, "drivers available.")
13  print("There will be", cars_not_driven, "empty cars today")
14  print("We can transport", carpool_capacity, "people today")
```

```
14 print("We can transport", carpool_capacity, "people today.")
15 print("We have", passengers, "to carpool today.")
16 print("We need to put about", average_passengers_per_car,
17       "in each car.")
```

---

## Info

The `_` in `space_in_a_car` is called an underscore character. Find out how to type it if you do not already know. We use this character a lot to put an imaginary space between words in variable names.

---

## What You Should See

```
1 There are 100 cars available.
2 There are only 30 drivers available.
3 There will be 70 empty cars today.
4 We can transport 120.0 people today.
5 We have 90 to carpool today.
6 We need to put about 3.0 in each car.
```

## Study Drills

When I wrote this program the first time, I had a mistake, and Python told me about it like this:

```
1 Traceback (most recent call last):
2   Cell In[1], line 8, in <module>
3       average_passengers_per_car = car_pool_capacity / passenge
4 NameError: name 'car_pool_capacity' is not defined
```

Explain this error in your own words. Make sure you use line numbers and explain why.

Here are more drills:

1. I used 4.0 for `space_in_a_car`, but is that necessary? What happens if it's just 4?
2. Remember that 4.0 is a floating point number. It's just a number with a decimal point, and you need 4.0 instead of just 4 so that it is floating point.
3. Write comments above each of the variable assignments.
4. Make sure you know what `=` is called (equals) and that its purpose is to give data (numbers, strings, etc.) names (`cars_driven`, `passengers`).
5. Remember that `_` is an underscore character.
6. Try running `python3` from the Terminal as a calculator like you did before, and use variable names to do your calculations. Popular variable names are also `i`, `x`, and `j`.

## Common Student Questions

**What is the difference between `=` (single-equal) and `==` (double-equal)?** The `=` (single-equal) assigns the value on the right to a variable on the left. The `==` (double-equal) tests whether two things have the same value. You'll learn about this later.

**Can we write `x=100` instead of `x = 100`?** You can, but it's bad form. You should add space around operators like this so that it's easier to read.

**What do you mean by “read the file (code) backward”?** Very simple. Imagine you have a file with 16 lines of code in it. Start at line 16, and compare it to my code at line 16. Then do it again for 15, and so on until you've read all of the code backward.

**Why did you use 4.0 for `space_in_a_car`?** It is mostly so you can then find out what a floating point number is and ask this question. See the *Study Drills* section.

## Exercise 5. More Variables and Printing

Now we'll do even more typing of variables and printing them out. This time we'll use something called a "format string." Every time you put " (double-quotes) around a piece of text you have been making a *string*. A string is how you make something that your program might give to a human. You print strings, save strings to files, send strings to web servers, and many other things.

Strings are really handy, so in this exercise you will learn how to make strings that have variables embedded in them. You embed variables inside a string by using a special {} sequence and then put the variable you want inside the {} characters. You also must start the string with the letter f for "format," as in `f"Hello {somevar}"`. This little f before the " (double-quote) and the {} characters tell Python 3, "Hey, this string needs to be formatted. Put these variables in there."

As usual, just type this in even if you do not understand it, and make it exactly the same.

Listing 5.1: ex5.py

```
1  my_name = 'Zed A. Shaw'
2  my_age = 35 # not a lie
3  my_height = 74 # inches
4  my_weight = 180 # lbs
5  my_eyes = 'Blue'
6  my_teeth = 'White'
7  my_hair = 'Brown'
8
9  print(f"Let's talk about {my_name}.")
10 print(f"He's {my_height} inches tall.")
11 print(f"He's {my_weight} pounds heavy.")
12 print("Actually that's not too heavy.")
13 print(f"Help, got {my_eyes} eyes and {my_hair} hair.")
```

```
13 print(f'He's got {my_eyes} eyes and {my_hair} hair. ')
14 print(f"His teeth are usually {my_teeth} depending on the coffee.")
15
16 # this line is tricky, try to get it exactly right
17 total = my_age + my_height + my_weight
18 print(f"If I add {my_age}, {my_height}, and {my_weight} I get {total}")
```

## What You Should See

```
1  Let's talk about Zed A. Shaw.
2  He's 74 inches tall.
3  He's 180 pounds heavy.
4  Actually that's not too heavy.
5  He's got Blue eyes and Brown hair.
6  His teeth are usually White depending on the coffee.
7  If I add 35, 74, and 180 I get 289.
```

## Study Drills

1. Change all the variables so there is no `my_` in front of each one. Make sure you change the name everywhere, not just where you used `=` to set them.
2. Try to write some variables that convert the inches and pounds to centimeters and kilograms. Do not just type in the measurements. Work out the math in Python.

## Common Student Questions

**Can I make a variable like this:** `1 = 'Zed Shaw'`? No, `1` is not a valid variable name. They need to start with a character, so `a1` would work, but `1` will not.

**How can I round a floating point number?** You can use the `round()` function like this: `round(1.7333)`.

**Why does this not make sense to me?** Try making the numbers in this script your measurements. It's weird, but talking about yourself will make it seem more real. Also, you're just starting out, so it won't make too much sense. Keep going and more exercises will explain it more.



## Exercise 6. Strings and Text

While you have been writing strings, you still do not know what they do. In this exercise we create a bunch of variables with complex strings so you can see what they are for. First an explanation of strings.

A string is usually a bit of text you want to display to someone or “export” out of the program you are writing. Python knows you want something to be a string when you put either " (double-quotes) or ' (single-quotes) around the text. You saw this many times with your use of `print` when you put the text you want to go inside the string inside " or ' after the `print` to print the string.

Strings can contain any number of variables that are in your Python script. Remember that a variable is any line of code where you set a name = (equal) to a value. In the code for this exercise, `types_of_people = 10` creates a variable named `types_of_people` and sets it = (equal) to 10. You can put that in any string with `{types_of_people}`. You also see that I have to use a special type of string to “format”; it’s called an “f-string” and looks like this:

```
f"some stuff here {avariable}"  
f"some other stuff {anothervar}"
```

Python *also* has another kind of formatting using the `.format()` syntax, which you see on line 17. You’ll see me use that sometimes when I want to apply a format to an already created string, such as in a loop. We’ll cover that more later.

We will now type in a whole bunch of strings, variables, and formats, and print them. You will also practice using short abbreviated variable names. Programmers love saving time at your expense by using annoyingly short and cryptic variable names, so let’s get you started reading and writing them early on.

## Listing 6.1: ex6.py

```
1  types_of_people = 10
2  x = f"There are {types_of_people} types of people."
3
4  binary = "binary"
5  do_not = "don't"
6  y = f"Those who know {binary} and those who {do_not}."
7
8  print(x)
9  print(y)
10
11 print(f"I said: {x}")
12 print(f"I also said: '{y}'")
13
14 hilarious = False
15 joke_evaluation = "Isn't that joke so funny?! {"
16
17 print(joke_evaluation.format(hilarious))
18
19 w = "This is the left side of..."
20 e = "a string with a right side."
21
22 print(w + e)
```

## What You Should See

```
1  There are 10 types of people.
2  Those who know binary and those who don't.
3  I said: There are 10 types of people.
4  I also said: 'Those who know binary and those who don't.'
5  Isn't that joke so funny?! False
6  This is the left side of...a string with a right side.
```

## Study Drills

1. Go through this program and write a comment above each line explaining it.
2. Find all the places where a string is put inside a string.
3. Are you sure there are only four places? How do you know? Maybe I like lying.
4. Explain why adding the two strings `w` and `e` with `+` makes a longer string.

## Break It

You are now at a point where you can try to break your code to see what happens. Think of this as a game to devise the most clever way to break the code. You can also find the simplest way to break it. Once you break the code, you then need to fix it. If you have a friend, then the two of you can try to break each other's code and fix it. Give your friend your code in a file named `ex6.py` so they can break something. Then you try to find their error and fix it. Have fun with this, and remember that if you wrote this code once, you can do it again. If you take your damage too far, you can always type it in again for extra practice.

## Common Student Questions

**Why do you put ' (single-quotes) around some strings and not others?** Mostly it's because of style, but I'll use a single-quote inside a string that has double-quotes. Look at lines 6 and 15 to see how I'm doing that.

**If you thought the joke was funny, could you write** `hilarious = True`**?** Yes, and you'll learn more about these boolean values later.

## Exercise 7. Combining Strings

Now we are going to do a bunch of exercises where you just type code in and make it run. I won't be explaining this exercise because it is more of the same. The purpose is to build up your chops. See you in a few exercises, and *do not skip!* Do not *paste!*

Listing 7.1: ex7.py

```
1  print("Mary had a little lamb.")
2  print("Its fleece was white as {}".format('snow'))
3  print("And everywhere that Mary went.")
4  print("." * 10) # what'd that do?
5
6  end1 = "C"
7  end2 = "h"
8  end3 = "e"
9  end4 = "e"
10 end5 = "s"
11 end6 = "e"
12 end7 = "B"
13 end8 = "u"
14 end9 = "r"
15 end10 = "g"
16 end11 = "e"
17 end12 = "r"
18
19 # watch end = ' ' at the end. try removing it to see what happens
20 print(end1 + end2 + end3 + end4 + end5 + end6, end=' ')
21 print(end7 + end8 + end9 + end10 + end11 + end12)
```

## What You Should See

```
1  Mary had a little lamb.  
2  Its fleece was white as snow.  
3  And everywhere that Mary went.  
4  .....  
5  Cheese Burger
```

## Study Drills

For these next few exercises, you will have the exact same Study Drills.

1. Go back through and write a comment on what each line does.
2. Read each one backward or out loud to find your errors.
3. From now on, when you make mistakes, write down on a piece of paper what kind of mistake you made.
4. When you go to the next exercise, look at the mistakes you have made and try not to make them in this new one.
5. Remember that everyone makes mistakes. Programmers are like magicians who fool everyone into thinking they are perfect and never wrong, but it's all an act. They make mistakes all the time.

## Break It

Did you have fun breaking the code in [Exercise 6](#)? From now on you're going to break all the code you write or a friend's code. I won't have a Break It section explicitly in every exercise, but I will do this in almost every video. Your goal is to find as many different ways to break your code until you get tired or exhaust all possibilities. In some exercises I might point out a specific common way people break that exercise's code, but otherwise consider this a standing order to always break it.

## Common Student Questions

**Why are you using the variable named 'snow'?** That's actually not a variable: it is just a string with the word snow in it. A variable wouldn't have the single-quotes around it.

**Is it normal to write an English comment for every line of code like you say to do in Study Drill 1?** No, you write comments only to explain difficult to understand code or why you did something. Why is usually much more important, and then you try to write the code so that it explains how something is being done on its own. However, sometimes you have to write such nasty code to solve a problem that it does need a comment on every line. In this case it's strictly for you to practice translating code to English.

**Can I use single-quotes or double-quotes to make a string or do they do different things?** In Python either way to make a string is acceptable, although typically you'll use single-quotes for any short strings like 'a' or 'snow'.

## Exercise 8. Formatting Strings Manually

We will now see how to do a more complicated formatting of a string. This code looks complex, but if you do your comments above each line and break each thing down to its parts, you'll understand it.

Listing 8.1: ex8.py

```
1  formatter = "{} {} {} {}"
2
3  print(formatter.format(1, 2, 3, 4))
4  print(formatter.format("one", "two", "three", "four"))
5  print(formatter.format(True, False, False, True))
6  print(formatter.format(formatter, formatter, formatter,
7  print(formatter.format(
8      "Try your",
9      "Own text here",
10     "Maybe a poem",
11     "Or a song about fear"
12 ))
```

### What You Should See

```
1  1 2 3 4
2  one two three four
3  True False False True
4  {} {} {} {} {} {} {} {} {} {} {} {} {} {} {} {}
5  Try your Own text here Maybe a poem Or a song about fear
```

In this exercise I'm using something called a "function" to turn the formatter variable into other strings.

When you see me write `formatter.format(...)`, I'm telling Python to do the following:

1. Take the `formatter` string defined on line 1.
2. Call its `format` function, which is similar to telling it to do a command line command named `format`.
3. Pass to `format` four arguments, which match up with the four `{}` in the `formatter` variable. This is like passing arguments to the command line command `format`.
4. The result of calling `format` on `formatter` is a new string that has the `{}` replaced with the four variables. This is what `print` is now printing out.

That's a lot for the eighth exercise, so what I want you to do is consider this a brainteaser. It's alright if you don't *really* understand what's going on because the rest of the book will slowly make this clear. At this point, try to study this and see what's going on, and then move on to the next exercise.

## Study Drills

Repeat the Study Drills from [Exercise 7](#).

## Common Student Questions

**Why do I have to put quotes around “one” but not around `True` or `False`?** Python recognizes `True` and `False` as keywords representing the concept of true and false. If you put quotes around them, then they are turned into strings and won't work. You'll learn more about how these work later.

**Can I use IDLE to run this?** No, you should use Jupyter or the command line if you know how. It is essential to learning programming and is a good place to start if you want to learn about programming. Jupyter is a far superior tool than IDLE.



## Exercise 9. Multi-line Strings

By now you should realize the pattern for this book is to use more than one exercise to teach you something new. I start with code that you might not understand, and then more exercises explain the concept. If you don't understand something now, you will later as you complete more exercises. Write down what you don't understand, and keep going.

Listing 9.1: ex9.py

```
1  # Here's some new strange stuff, remember type it exact
2
3  days = "Mon Tue Wed Thu Fri Sat Sun"
4  months = "Jan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"
5
6  print("Here are the days: ", days)
7  print("Here are the months: ", months)
8
9  print("""
10 There's something going on here.
11 With the three double-quotes.
12 We'll be able to type as much as we like.
13 Even 4 lines if we want, or 5, or 6.
14 """)
```

## What You Should See

```
1 Here are the days: Mon Tue Wed Thu Fri Sat Sun
2 Here are the months: Jan
3 Feb
4 Mar
5 Apr
6 May
```

```
7   Jun
8   Jul
9   Aug
10
11  There's something going on here.
12  With the three double-quotes.
13  We'll be able to type as much as we like.
14  Even 4 lines if we want, or 5, or 6.
```

## Study Drills

Repeat the Study Drills from [Exercise 7](#).

## Common Student Questions

**Why do I get an error when I put spaces between the three double-quotes?** You have to type them like `"""` and not `" "`, meaning with *no* spaces between each one.

**What if I wanted to start the months on a new line?** You simply start the string with `\n` like this:  
`"\nJan\nFeb\nMar\nApr\nMay\nJun\nJul\nAug"`.

**Is it bad that my errors are always spelling mistakes?** Most programming errors in the beginning (and even later) are simple spelling mistakes, typos, or getting simple things out of order.

## Exercise 10. Escape Codes in Strings

In [Exercise 9](#) I threw you some new stuff just to keep you on your toes. I showed you two ways to make a string that goes across multiple lines. In the first way, I put the characters `\n` (backslash n) between the names of the months. These two characters put a new line character into the string at that point.

This `\` (backslash) character encodes difficult-to-type characters into a string. There are various “escape sequences” available for different characters you might want to use. We’ll try a few of these sequences so you can see what I mean.

An important escape sequence is to escape a single-quote `'` or double-quote `"`. Imagine you have a string that uses double-quotes and you want to put a double-quote inside the string. If you write `"I "understand" joe."` then Python will get confused because it will think the `"` around `"understand"` actually *ends* the string. You need a way to tell Python that the `"` inside the string isn’t a *real* double-quote.

To solve this problem you *escape* double-quotes and single-quotes so Python knows to include them in the string. Here’s an example:

```
"I am 6'2\" tall." # escape double-quote inside string
'I am 6\'2" tall.' # escape single-quote inside string
```

The second way to solve this problem is to use triple-quotes, which is just `"""` and works like a string, but you also can put as many lines of text as you want until you type `"""` again. We’ll also play with these.

Listing 10.1: ex10.py

---

```
1 tabby_cat = "\tI'm tabbed in."
2 persian_cat = "I'm split\non a line."
3 backslash_cat = "I'm \\ a \\ cat."
```

```
4
5 fat_cat = """
6 I'll do a list:
7 \t* Cat food
8 \t* Fishies
9 \t* Catnip\n\t* Grass
10 """
11
12 print(tabby_cat)
13 print(persian_cat)
14 print(backslash_cat)
15 print(fat_cat)
```

## What You Should See

Look for the tab characters that you made. In this exercise the spacing is important to get right.

```
1          I'm tabbed in.
2 I'm split
3 on a line.
4 I'm \ a \ cat.
5
6 I'll do a list:
7         * Cat food
8         * Fishies
9         * Catnip
10        * Grass
```

## Escape Sequences

This is all of the escape sequences Python supports. You may not use many of these, but memorize their format and what they do anyway. Try them out in some strings to see if you can make them work.

Escape	What it does.
\\	Backslash (\)
\'	Single-quote (')
\"	Double-quote (")
\a	ASCII bell (BEL)
\b	ASCII backspace (BS)
\f	ASCII formfeed (FF)
\n	ASCII linefeed (LF)
\N{name}	Character named name in the Unicode database (Unicode only)
\r	Carriage return (CR)
\t	Horizontal tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx
\v	ASCII vertical tab (VT)
\000	Character with octal value 000
\xhh	Character with hex value hh

## Study Drills

1. Memorize all the escape sequences by putting them on flash cards.
2. Use ''' (triple-single-quote) instead. Can you see why you might use that instead of ""?"
3. Combine escape sequences and format strings to create a more complex format.

## Common Student Questions

**I still haven't completely figured out the last exercise. Should I continue?** Yes, keep going. Instead of stopping, take notes listing things you don't understand for each exercise. Periodically go

through your notes and see if you can figure these things out after you've completed more exercises. Sometimes, though, you may need to go back a few exercises and do them again.

**What makes `//` special compared to the other ones?** It's simply the way you would write out one backslash (`\`) character. Think about why you would need this.

**When I write `//` or `/n` it doesn't work.** That's because you are using a forward-slash `/` and not a backslash `\`. They are different characters that do very different things.

**I don't get Study Drill 3. What do you mean by "combine" escape sequences and formats?** One concept I need you to understand is that each of these exercises can be combined to solve problems. Take what you know about format strings and write some new code that uses format strings *and* the escape sequences from this exercise.

**What's better, `'''` or `"""`?** It's entirely based on style. Go with the `'''` (triple-single-quote) style for now, but be ready to use either depending on what feels best or what everyone else is doing.

## Exercise 11. Asking People Questions

Now it is time to pick up the pace. You are doing a lot of printing to get you familiar with typing simple things, but those simple things are fairly boring. What we want to do now is get data into your programs. This is a little tricky because you have to learn to do two things that may not make sense right away, but trust me and do them anyway. They will make sense in a few exercises.

Most of what software does is the following:

1. Take some kind of input from a person.
2. Change it.
3. Print out something to show how it changed.

So far you have been printing strings, but you haven't been able to get any input from a person. You may not even know what "input" means, but type this code in anyway and make it exactly the same. In the next exercise we'll do more to explain input.

Listing 11.1: ex11.py

---

```
1  print("How old are you?", end=' ')
2  age = input()
3  print("How tall are you?", end=' ')
4  height = input()
5  print("How much do you weigh?", end=' ')
6  weight = input()
7
8  print(f"So, you're {age} old, {height} tall and {weight}")
```

---

### Info

We put an `end=' '` at the end of each `print` line. This tells `print` to not end the line with a newline character and go to the next line.

---

## What You Should See

```
1  How old are you? 38
2  How tall are you? 6'2"
3  How much do you weigh? 180lbs
4  So, you're 38 old, 6'2" tall and 180lbs heavy.
```

## Study Drills

1. Go online and find out what Python's `input` does.
2. Can you find other ways to use it? Try some of the samples you find.
3. Write another “form” like this to ask some other questions.

## Common Student Questions

**How do I get a number from someone so I can do math?** That's a little advanced, but try `x = int(input())`, which gets the number as a string from `input()` and then converts it to an integer using `int()`.

**I put my height into raw input using `input("6'2")` but it doesn't work.** You don't put your height in there; you type it directly into your Terminal. First thing is, go back and make the code exactly like mine. Next, run the script, and when it pauses, type your height in at your keyboard. That's all there is to it.



## Exercise 12. An Easier Way to Prompt

When you typed `input()`, you were typing the `(` and `)` characters, which are parenthesis characters. For `input` you can also put in a prompt to show to a person so they know what to type. Put a string that you want for the prompt inside the `()` so that it looks like this:

```
y = input("Name? ")
```

This prompts the user with “Name?” and puts the result into the variable `y`. This is how you ask someone a question and get the answer.

This means we can completely rewrite our previous exercise using just `input` to do all the prompting.

Listing 12.1: `ex12.py`

```
1 age = input("How old are you? ")
2 height = input("How tall are you? ")
3 weight = input("How much do you weigh? ")
4
5 print(f"So, you're {age} old, {height} tall and {weight}
```

## What You Should See

```
1 How old are you? 38
2 How tall are you? 6'2"
3 How much do you weigh? 180lbs
4 So, you're 38 old, 6'2" tall and 180lbs heavy.
```

## Study Drills

1. In your Jupyter cell right-click on any print and select Show Contextual Help. This will give you quick documentation for print.
2. If you do this and the panel says “Click on a function to see documentation.” then you need to run the code with SHIFT-ENTER and then click on the print function again.
3. Next, go to [Google](#) and search for python print site:python.org to get official documentation for the Python print function.

## Common Student Questions

**Why does the Contextual Help disappear?** I’m not sure, but I suspect it can’t figure out what function you want documentation for while you’re editing the code. Run the code and then suddenly it’ll work. You can also click on any other functions in any other cell you work on.

**Where does this documentation come from?** These are documentation comments added to the code itself, which is why it might be different from the online documentation. Get in the habit of studying both when you can.

## Exercise 13. Parameters, Unpacking, Variables

We're now going to take a quick detour into the world of the `Terminal` (aka `PowerShell`) version of `python`. If you did the [First Steps](#) correctly, you should have learned how to start your `Terminal` and run a simple `Python` script. Later in this course you'll learn how to use the `Terminal` more extensively, but in this exercise we'll just do a tiny test.

First, I want you to create a file named `ex13.py` using Jupyter's new `Python` file:

1. On the left there's a list of the files in your directory.
2. Above that list is a blue `[+]` button.
3. Click that button, and scroll to the very bottom where there should be a button for `Python File` with the `Python` "blue and yellow snakes" logo.
4. Click that button to open a new panel you can type code into.
5. Right away, use your mouse to select `File > Save Python File` or hold `CTRL` and hit `s` (normally shown as `Ctrl-S` but you don't use `shift` to get that `s`).
6. This will open a modal prompt that says "Rename File." Type "`ex13`" and it should keep the `.py`, but be sure that this input says `ex13.py`.
7. Hit the blue `[Rename]` button to save the file in that directory.

Once that file is saved, you can then type this code into the file:

Listing 13.1: `ex13.py`

---

```
1  from sys import argv
2  # read the What You Should See section for how to run this
3  script, first, second, third = argv
4
```

```
4
5 print("The script is called:", script)
6 print("Your first variable is:", first)
7 print("Your second variable is:", second)
8 print("Your third variable is:", third)
```

I recommend you type only one or two lines of code and then do the following:

1. Save your file again. CTRL-s is the easiest way, but use the menu if you can't remember it. This time it shouldn't ask you to "rename" the file but instead should just save it.
2. Your file is now saved to your projects directory. If you remember from the *First Steps* section, you created a directory in `~/Projects/lpythw`, and when you run `jupyter-lab`, you first `cd ~/Projects/lpythw`.
3. Now start a new Terminal (aka PowerShell on Windows) and `cd ~/Projects/lpythw/` again to get a Terminal there.
4. Finally, type `python ex13.py first 2nd 3rd`. (Type this without the Terminal period.) When you do, you should see *absolutely nothing*! Yes, this is *very important*. You only typed one or two lines, so there are no print lines in your code. That means it does not print anything, but that's good. If you get errors, then stop and figure out what you're doing wrong. Did you type that line wrong? Did you run `python ex13.py`? That is also wrong. You have to run `python ex13.py first 2nd 3rd`. (Again, type without the Terminal period.)

## If You Get Lost

If you're confused about where you are, use the `open` command on macOS and the `start` command on Windows. If you type this:

```
1 open .
```

on a macOS computer, it will open a window with the contents of where your Terminal is currently located. The same happens when you type:

```
1 start .
```



on Windows inside PowerShell. Doing this will help you connect your idea of “files are in folders in a window” to “files are in directories in the Terminal (PowerShell).”

If this is the first time you’re seeing this advice, then go back to the [First Steps](#) section and review it as it seems you missed this important concept.

## Code Description

On line 1 we have what’s called an “import.” This is how you add features to your script from the Python feature set. Rather than give you all the features at once, Python asks you to say what you plan to use. This keeps your programs small, but it also acts as documentation for other programmers who read your code later.

The `argv` is the “argument variable,” a very standard name in programming that you will find used in many other languages. This variable *holds* the arguments you pass to your Python script when you run it. In the exercises you will get to play with this more and see what happens.

Line 3 “unpacks” `argv` so that, rather than holding all the arguments, it gets assigned to four variables you can work with: `script`, `first`, `second`, and `third`. This may look strange, but “unpack” is probably the best word to describe what it does. It just says, “Take whatever is in `argv`, unpack it, and assign it to all of these variables on the left in order.”

After that we just print them out like normal.

## Hold Up! Features Have Another Name

I call them “features” here (these little things you `import` to make your Python program do more), but nobody else calls them features. I just used

that name because I needed to trick you into learning what they are without jargon. Before you can continue, you need to learn their real name: `modules`.

From now on we will be calling these “features” that we import *modules*. I’ll say things like, “You want to import the `sys` module.” They are also called “libraries” by other programmers, but let’s just stick with modules.

## What You Should See

---

### Warning!

Pay attention! You have been running Python scripts without command line arguments. If you type only `python3 ex13.py`, you are doing it wrong! Pay close attention to how I run it. This applies any time you see `argv` being used.

---

When you are done typing in all of the code, it should finally run like this (and you *must* pass *three* command line arguments):

```
1  $ python ex13.py first 2nd 3rd
2  The script is called: ex13.py
3  Your first variable is: first
4  Your second variable is: 2nd
5  Your third variable is: 3rd
```

This is what you should see when you do a few different runs with different arguments:

```
1  $ python ex13.py stuff things that
2  The script is called: ex13.py
3  Your first variable is: stuff
4  Your second variable is: things
5  Your third variable is: that
```

Here’s one more example showing it can be anything:

```
1 $ python ex13.py apple orange grapefruit
2 The script is called: ex13.py
3 Your first variable is: apple
4 Your second variable is: orange
5 Your third variable is: grapefruit
```

You can actually replace `first`, `2nd`, and `3rd` with any three things you want.

If you do not run it correctly, then you will get an error like this:

```
Command failed: python ex13.py first 2nd
Traceback (most recent call last):
  File "/Users/zedshaw/Project
s/learncodethehardway.com/private/db/modules/learn-python-the-hard-way-
5e-section-1/code/ex13.py", line 3, in script, first, second, third = argv
ValueError: not enough values to unpack (expected 4, got 3)
```

This happens when you do not put enough arguments on the command when you run it (in this case just `first 2nd`). Notice that when I run it, I give it `first 2nd`, which caused it to give an error about “need more than 3 values to unpack” telling you that you didn’t give it enough parameters.

## Study Drills

1. Try giving fewer than three arguments to your script. See that error you get? See if you can explain it.
2. Write a script that has fewer arguments and one that has more. Make sure you give the unpacked variables good names.
3. Combine `input` with `argv` to make a script that gets more input from a user. Don’t overthink it. Just use `argv` to get something, and use `input` to get something else from the user.
4. Remember that modules give you features. Modules. Modules. Remember this because we’ll need it later.

## Common Student Questions

**When I run it, I get** `ValueError: need more than 1 value to unpack`. Remember that an important skill is paying attention to details. If you look at the *What You Should See* section, you see that I run the script with parameters on the command line. You should replicate how I ran it exactly. There's also a giant warning right there explaining the mistake you just made, so again, please pay attention.

**What's the difference between** `argv` **and** `input()`? The difference has to do with where the user is required to give input. If they give your script inputs on the command line, then you use `argv`. If you want them to input using the keyboard while the script is running, then use `input()`.

**Are the command line arguments strings?** Yes, they come in as strings, even if you typed numbers on the command line. Use `int()` to convert them just like with `int(input())`.

**How do you use the command line?** You should have learned to use it very quickly and fluently by now, but if you need to learn it at this stage, then read Appendix A, "Command Line Crash Course."

**I can't combine** `argv` **with** `input()`. Don't overthink it. Just slap two lines at the end of this script that uses `input()` to get something and then print it. From that, start playing with more ways to use both in the same script.

**Why can't I do this** `input('? ') = x`? Because that's backward to how it should work. Do it the way I do it and it'll work.

**Why do you want me to type one line at a time?** The biggest mistake beginners make—and professionals too—is they type a massive block of code, run it once, and then cry because of all the errors they have to fix. Errors in programming languages are awful and frequently point at the wrong locations in your source. If you're typing only a few lines at a time, you will run your code more often, and when you get an error, you know it's probably a problem with the line(s) you just typed. When you type 100 lines of code, you'll spend the next 5 days trying to find all the errors and just give up.



Save yourself the trouble and just type a little at a time. It's what I—  
and most capable programmers—do in real life.

## Exercise 14. Prompting and Passing

Let's do an exercise that uses `argv` and `input` together to ask the user something specific. You will need this for the next exercise where you learn to read and write files. In this exercise we'll use `input` slightly differently by having it print a simple `>` prompt.

Listing 14.1: `ex14.py`

```
1  from sys import argv
2
3  script, user_name = argv
4  prompt = '> '
5
6  print(f"Hi {user_name}, I'm the {script} script.")
7  print("I'd like to ask you a few questions.")
8  print(f"Do you like me {user_name}?")
9  likes = input(prompt)
10
11 print(f"Where do you live {user_name}?")
12 lives = input(prompt)
13
14 print("What kind of computer do you have?")
15 computer = input(prompt)
16
17 print(f"""
18 Alright, so you said {likes} about liking me.
19 You live in {lives}. Not sure where that is.
20 And you have a {computer} computer. Nice.
21 """)
```

We make a variable `prompt` that is set to the prompt we want, and we give that to `input` instead of typing it over and over. Now if we want to make the

prompt something else, we just change it in this one spot and rerun the script. Very handy.

---

## Warning!

Remember that you have to do the same thing you did in [Exercise 13](#) and use the Terminal to make this work. This will be the last time for a while, but it's important to know how to run code from the Terminal since that's a very common way to run Python code.

---

## What You Should See

When you run this, remember that you have to give the script your name for the argv arguments.

```
1  $ python ex14.py zed
2  Hi zed, I'm the ex14.py script.
3  I'd like to ask you a few questions.
4  Do you like me zed?
5  > Yes
6  Where do you live zed?
7  > San Francisco
8  What kind of computer do you have?
9  > Tandy 1000
10
11 Alright, so you said Yes about liking me.
12 You live in San Francisco. Not sure where that is.
13 And you have a Tandy 1000 computer. Nice.
```

## Study Drills

1. Find out what the games Zork and Adventure were. Try to find a copy and play them.
2. Change the prompt variable to something else entirely.

3. Add another argument and use it in your script, the same way you did in the previous exercise with `first, second = ARGV`.
4. Make sure you understand how I combined a `"""` style multi-line string with the `{}` format activator as the last print.
5. Try to find a way to run this in Jupyter. You will probably have to replace the code that uses `argv` with something else, like some variables.

## Common Student Questions

**I get `SyntaxError: invalid syntax` when I run this script.** Again, you have to run it right on the command line, not inside Python. If you type `python3` and then try to type `python3 ex14.py Zed`, it will fail because you are running *Python inside Python*. Close your window and then just type `python3 ex14.py Zed`.

**I don't understand what you mean by changing the prompt.** See the variable `prompt = '> '`. Change that to have a different value. You know this; it's just a string and you've done 13 exercises making them, so take the time to figure it out.

**I get the error `ValueError: need more than 1 value to unpack`.** Remember when I said you need to look at the *What You Should See* (WYSS) section and replicate what I did? You need to do the same thing here and focus on how I type the command in and why I have a command line argument.

**How can I run this from IDLE?** Don't use IDLE. It's garbage.

**Can I use double-quotes for the prompt variable?** You totally can. Go ahead and try that.

**You have a Tandy computer?** I did when I was little.

**I get `NameError: name 'prompt' is not defined` when I run it.** You either spelled the name of the prompt variable wrong or forgot that line. Go back and compare each line of code to mine, from at the

bottom of the script to the top. Any time you see this error, it means you spelled something wrong or forgot to create the variable.

## Exercise 15. Reading Files

You know how to get input from a user with `input` or `argv`. Now you will learn about reading from a file. You may have to play with this exercise the most to understand what's going on, so do the exercise carefully and remember your checks. Working with files is an easy way to *erase your work* if you are not careful.

This exercise involves writing two files. One is the usual `ex15.py` file that you will run, but the *other* is named `ex15_sample.txt`. This second file isn't a script but a plain-text file we'll be reading in our script. Here are the contents of that file:

```
1  This is stuff I typed into a file.  
2  It is really cool stuff.  
3  Lots and lots of fun to have in here.
```

What we want to do is “open” that file in our script and print it out. However, we do not want to just “hard-code” the name `ex15_sample.txt` into our script. “Hard-coding” means putting some bit of information that should come from the user as a string directly in our source code. That's bad because we want it to load other files later. The solution is to use `argv` or `input` to ask the user what file to open instead of “hard-coding” the file's name.

Listing 15.1: `ex15.py`

---

```
1  from sys import argv  
2  
3  script, filename = argv  
4  
5  txt = open(filename)  
6  
7  print(f"Here's your file {filename}:")
```

```
8  print(txt.read())
9
10 print("Type the filename again:")
11 file_again = input("> ")
12
13 txt_again = open(file_again)
14
15 print(txt_again.read())
```

A few fancy things are going on in this file, so let's break it down really quickly.

Lines 1–3 uses `argv` to get a filename. Next we have line 5, where we use a new command: `open`. Right now, run `pydoc open` and read the instructions. Notice how, like your own scripts and `input`, it takes a parameter and returns a value you can set to your own variable. You just opened a file.

Line 7 prints a little message, but on line 8 we have something very new and exciting. We call a function on `txt` named `read`. What you get back from `open` is a *file*, and it also has commands you can give it. You give a file a command by using the `.` (dot or period), the name of the command, and parameters, just like with `open` and `input`. The difference is that `txt.read()` says, “Hey `txt`! Do your `read` command with no parameters!”

The remainder of the file is more of the same, but we'll leave the analysis to you in the *Study Drills* section.

## What You Should See

---

### Warning!

Pay attention! I said pay attention! You have been running scripts with just the name of the script, but now that you are using `argv` you have to add arguments. Look at the very first line of the following example, and you will see I do `python ex15.py ex15_sample.txt` to run it. See the extra argument `ex15_sample.txt` after the `ex15.py`

script name. If you do not type that, you will get an error, so pay attention!

---

I made a file called `ex15_sample.txt` and ran my script.

```
1  Here's your file ex15_sample.txt:
2  This is stuff I typed into a file.
3  It is really cool stuff.
4  Lots and lots of fun to have in here.
5
6
7  Type the filename again:
8  > ex15_sample.txt
9  This is stuff I typed into a file.
10 It is really cool stuff.
11 Lots and lots of fun to have in here.
```

## Study Drills

This is a big jump, so be sure you do these Study Drills as best you can before moving on.

1. Above each line, write out in English what that line does.
2. If you are not sure, ask someone for help or search online. Many times searching for “python3 THING” will find answers to what that THING does in Python. Try searching for “python3 open.”
3. I used the word “commands” here, but commands are also called “functions” and “methods.” You will learn about functions and methods later in the book.
4. Get rid of the lines 10–15 where you use `input` and run the script again.
5. Use only `input` and try the script that way. Why is one way of getting the filename better than another?



6. Start `python3` to start the `python3` shell, and use `open` from the prompt just like in this program. Notice how you can open files and run `read` on them from within `python3`?
7. Have your script also call `close()` on the `txt` and `txt_again` variables. It's important to close files when you are done with them.

## Common Student Questions

**Does `txt = open(filename)` return the contents of the file?** No, it doesn't. It actually makes something called a "file object." You can think of a file like an old tape drive that you saw on mainframe computers in the 1950s or even like a DVD player from today. You can move around inside them and then "read" them, but the DVD player is not the DVD the same way the file object is not the file's contents.

**I can't type code into Terminal/PowerShell like you say in Study Drill 7.** First thing, from the command line just type `python3` and press Enter. Now you are in `python3` as we've done a few other times. Then you can type in code and Python will run it in little pieces. Play with that. To get out of it type `quit()` and hit Enter.

**Why is there no error when we open the file twice?** Python will not restrict you from opening a file more than once, and sometimes this is necessary.

**What does `from sys import argv` mean?** For now just understand that `sys` is a package, and this phrase just says to get the `argv` feature from that package. You'll learn more about this later.

**I put the name of the file in as `script, ex15_sample.txt = argv`, but it doesn't work.** No, that's not how you do it. Make the code exactly like mine, and then run it from the command line the exact same way I do. You don't put the names of files in; you let Python put the name in.

## Exercise 16. Reading and Writing Files

If you did the Study Drills from the previous exercise, you should have seen all sorts of commands (methods/functions) you can give to files. Here's the list of commands I want you to remember:

- `close` – Closes the file. Like `File->Save..` in a text editor or word processor.
- `read` – Reads the contents of the file. You can assign the result to a variable.
- `readline` – Reads just one line of a text file.
- `truncate` – Empties the file. Watch out if you care about the file.
- `write('stuff')` – Writes “stuff” to the file.
- `seek(0)` – Move the read/write location to the beginning of the file.

One way to remember what each of these does is to think of a vinyl record, cassette tape, VHS tape, DVD, or CD player. In the early days of computers data was stored on each of these kinds of media, so many of the file operations still resemble a storage system that is linear. Tape and DVD drives need to “seek” a specific spot, and then you can read or write at that spot. Today we have operating systems and storage media that blur the lines between random access memory and disk drives, but we still use the older idea of a linear tape with a read/write head that must be moved.

For now, these are the important commands you need to know. Some of them take parameters, but we do not really care about that. You only need to remember that `write` takes a parameter of a string you want to write to the file.

Let's use some of this to make a simple little text editor:

Listing 16.1: `ex16.py`

---

```
1 filename = "test.txt"
2
3 print(f"We're going to erase {filename}.")
4 print("If you don't want that, hit CTRL-C (*^C*).")
5 print("If you do want that, hit RETURN.")
6
7 input("?")
8
9 print("Opening the file...")
10 target = open(filename, 'w')
11
12 print("Truncating the file. Goodbye!")
13 target.truncate()
14
15 print("Now I'm going to ask you for three lines.")
16
17 line1 = input("line 1: ")
18 line2 = input("line 2: ")
19 line3 = input("line 3: ")
20
21 print("I'm going to write these to the file.")
22
23 target.write(line1)
24 target.write("\n")
25 target.write(line2)
26 target.write("\n")
27 target.write(line3)
28 target.write("\n")
29
30 print("And finally, we close it.")
31 target.close()
```

That's a large file, probably the largest you have typed in. So go slow, do your checks, *run it frequently*, and take it slowly. One trick is to get bits of it running at a time. Get lines 1–2 running, then two more, then a few more, until it's all done and running.

## What You Should See

There are actually two things you will see. First the output of your new script:

```
1  We're going to erase test.txt.
2  If you don't want that, hit CTRL-C (^C).
3  If you do want that, hit RETURN.
4  ?
5  Opening the file...
6  Truncating the file. Goodbye!
7  Now I'm going to ask you for three lines.
8  line 1:  Mary had a little lamb
9  line 2:  Its fleece was white as snow
10 line 3:  It was also tasty
11 I'm going to write these to the file.
12 And finally, we close it.
```

Now, open up the file you made (in my case `test.txt`) using the left panel of Jupyter and check it out. Neat, right?

## Study Drills

1. If you do not understand this, go back through and use the comment trick to get it squared away in your mind. One simple English comment above each line will help you understand or at least let you know what you need to research more.
2. Write a `.py` script similar to the last [Exercise 14](#) that uses `read` ([Exercise 15](#)) and `argv` ([Exercise 13](#)) to read the file you just created. Be sure you run this in Terminal/PowerShell instead of Jupyter.
3. There's too much repetition in this file. Use strings, formats, and escapes to print out `line1`, `line2`, and `line3` with just one `target.write()` command instead of six.
4. Find out why we had to pass a `'w'` as an extra parameter to `open`. Hint: `open` tries to be safe by making you explicitly say you want to

write a file.

5. If you open the file with 'w' mode, then do you really need the `target.truncate()`? Read the documentation for Python's `open` function and see if that's true.

## Common Student Questions

**Is the `truncate()` necessary with the 'w' parameter?** See Study Drill 5.

**What does 'w' mean?** It's really just a string with a character in it for the kind of mode for the file. If you use 'w', then you're saying "open this file in 'write' mode," which is the reason for the 'w' character. There's also 'r' for "read," 'a' for append, and modifiers on these.

**What modifiers to the file modes can I use?** The most important one to know for now is the + modifier, so you can do 'w+', 'r+', and 'a+'. This will open the file in both read and write mode and, depending on the character use, position the file in different ways.

**Does just doing `open(filename)` open it in 'r' (read) mode?** Yes, that's the default for the `open()` function.

## Exercise 17. More Files

Now let's do a few more things with files. We'll write a Python script to copy one file to another. It'll be very short but will give you ideas about other things you can do with files.

Listing 17.1: ex17.py

```
1  from sys import argv
2  from os.path import exists
3
4  from_file = "test.txt"
5  to_file = "new_test.txt"
6
7  print(f"Copying from {from_file} to {to_file}")
8
9  # we could do these two on one line, how?
10 in_file = open(from_file)
11 indata = in_file.read()
12
13 print(f"The input file is {len(*indata)*} bytes long")
14
15 print(f"Does the output file exist? {exists(*to_file)}")
16 print("Ready, hit RETURN to continue, CTRL-C to abort.")
17 input()
18
19 out_file = open(to_file, 'w')
20 out_file.write(indata)
21
22 print("Alright, all done.")
23
24 out_file.close()
25 in_file.close()
```

You should immediately notice that we import another handy command named `exists`. This returns `True` if a file exists, based on its name in a string as an argument. It returns `False` if not. We'll be using this function in the second half of this book to do lots of things, but right now you should see how you can import it.

Using `import` is a way to get a ton of free code other better (well, usually) programmers have written so you do not have to write it.

## What You Should See

Just like your other scripts, run this one with two arguments: the file to copy from and the file to copy it to. I'm going to use a simple test file named `test.txt`:

```
1 Copying from test.txt to new_test.txt
2 The input file is 70 bytes long
3 Does the output file exist? True
4 Ready, hit RETURN to continue, CTRL-C to abort.
5
6 Alright, all done.
```

It should work with any file. Try a bunch more and see what happens. Just be careful you do not blast an important file.

---

### Warning!

Did you see the trick I did with `echo` to make a file and `cat` to show the file? You can learn how to do that in Appendix A, “Command Line Crash Course.”

---

## Study Drills

1. This script is *really* annoying. There's no need to ask you before doing the copy, and it prints too much out to the screen. Try to make

the script friendlier to use by removing features.

2. See how short you can make the script. I could make this one line long.
3. Notice at the end of What You Should See I used something called `cat`? It's an old command that "concatenates" files together, but mostly it's just an easy way to print a file to the screen. Type `man cat` to read about it.
4. Find out why you had to write `out_file.close()` in the code.
5. Go read up on Python's `import` statement, and start `python3` to try it out. Try importing some things and see if you can get it right. It's alright if you do not.
6. Try converting this code to an `ex17.py` script you can run from Terminal/PowerShell again. If you're getting tired of Jupyter's text editor, then check out one of the editors mentioned in the [First Steps](#) section.

## Common Student Questions

**Why is the 'w' in quotes?** That's a string. You've been using strings for a while now. Make sure you know what a string is.

**No way you can make this one line!** That ; depends ; on ; how ; you ; define ; one ; line ; of ; code.

**Is it normal to feel like this exercise was really hard?** Yes, it is totally normal. Programming may not "click" for you until maybe even [Exercise 36](#), or it might not until you finish the book and then make something with Python. Everyone is different, so just keep going and keep reviewing exercises that you had trouble with until it clicks. Be patient.

**What does the `len()` function do?** It gets the length of the string that you pass to it and then returns that as a number. Play with it.



**When I try to make this script shorter, I get an error when I close the files at the end.** You probably did something like this, `indata = open(from_file).read()`, which means you don't need to then do `in_file.close()` when you reach the end of the script. It should already be closed by Python once that one line runs.

**I get a Syntax:EOL while scanning string literal error.** You forgot to end a string properly with a quote. Go look at that line again.

## **Module 2: The Basics of Programming**

## Exercise 18. Names, Variables, Code, Functions

Big title, right? I am about to introduce you to *the function*! Dum dum dah! Every programmer will go on and on about functions and all the different ideas about how they work and what they do, but I will give you the simplest explanation you can use right now.

Functions do three things:


1. They name pieces of code the way variables name strings and numbers
2. They take arguments the way Python scripts take argv in [Exercise 13](#)
3. Using 1 and 2, they let you make your own “mini-scripts” or “tiny commands”

You can create an *empty* function by using the word `def` in Python like this:

Listing 18.1: ex18\_demo.py

---

```
1  def do_nothing():
2      pass
```




This creates the function, but the `pass` keyword tells Python this function is empty. To make the function do something, you add the code for the function *under* the `def` line, but indent it four spaces:

Listing 18.2: ex18\_demo.py

---

```
1  def do_something():
2      print("I did something!")
```




This effectively assigns the code `print("I did something!")` to the name `do_something` so you can then use it again later in your code, similar to

other variables. Using a function you’ve defined is how you “run” it, or “call” it:


Listing 18.3: ex18\_demo.py

---

```
1  def do_something():
2      print("I did something!")
3
4  # now we can call it by its name
5  do_something()
```



When the `do_something()` at the bottom runs, Python does the following:


1. Finds the `do_something` function in Python’s memory
2. Sees you’re calling it with `()`
3. Jumps to where the `def do_something()` line is
4. Runs the lines of code *under* the `def`, which in this case is one line:  
`print("I did`
5.  something!" When the code under the `def` is finished, Python exits the function and jumps back to where you called it
6. Then it continues, which in this case is the end of the code

For this exercise you need only one more concept, which is “arguments” to functions:

Listing 18.4: ex18\_demo.py

---

```
1  def do_more_things(a, b):
2      print("A IS", a, "B IS", b)
3
4  do_more_things("hello", 1)
```



In this case, I have two arguments (also called “parameters”) to the `do_more_things` function: `a` and `b`. When I call this function using `do_more_things("hello", 1)`, Python *temporarily* assigns `a="hello"` and `b=1` and then calls the function. That means, inside the function `a` and `b` will have those values, and they’ll disappear when the function exits. It’s kind of like doing this:

Listing 18.5: `ex18_demo.py`

```
1  def do_more_things(a, b):
2      a = "hello"
3      b = 1
4      print("A IS", a, "B IS", b)
```

Keep in mind this is not entirely accurate, since if you called `do_more_things` with different arguments, the `a` and `b` would be different. It’s only an example of this *one* time you call it with `do_more_things("hello", 1)`.

## Exercise Code

Take some time right now to play around in Jupyter by making your own functions and calling them before attempting this code. Be sure you understand how your code jumps to functions and then jumps back. Then I’m going to have you make four different functions, and I’ll then show you how each one is related:

Listing 18.6: `ex18.py`

```
1  # this one is like your scripts with argv
2  def print_two(*args):
3      arg1, arg2 = args
4      print(f"arg1: {arg1}, arg2: {arg2}")
5
6  # ok, that *args is actually pointless, we can just do this
7  def print_two_again(arg1, arg2):
8      print(f"arg1: {arg1}, arg2: {arg2}")
```

```

8         print(' arg1: {arg1}, arg2: {arg2} ')
9
10 # this just takes one argument
11 def print_one(arg1):
12     print(f"arg1: {arg1}")
13
14 # this one takes no arguments
15 def print_none():
16     print("I got nothin'.")
17
18
19 print_two("Zed", "Shaw")
20 print_two_again("Zed", "Shaw")
21 print_one("First!")
22 print_none()

```

Let's break down the first function, `print_two`, which is the most similar to what you already know from making scripts:

1. First we tell Python we want to make a function using `def` for "define".
2. On the same line as `def` we give the function a name. In this case we just called it "`print_two`", but it could also be "`peanuts`". It doesn't matter, except that your function should have a short name that says what it does.
3. Then we tell it we want `*args` (asterisk args), which is a lot like your `argv` parameter but for functions. This *has* to go inside `()` parentheses to work.
4. Then we end this line with a `:` (colon) and start indenting.
5. After the colon all the lines that are indented four spaces will become attached to this name, `print_two`. Our first indented line is one that unpacks the arguments, the same as with your scripts.
6. To demonstrate how it works we print these arguments out, just like we would in a script.

The problem with `print_two` is that it's not the easiest way to make a function. In Python we can skip the whole unpacking arguments and just use the names we want right inside `()`. That's what `print_two_again` does.

After that you have an example of how you make a function that takes one argument in `print_one`.

Finally you have a function that has no arguments in `print_none`.

---

## Warning!

This is very important. Do not get discouraged right now if this doesn't quite make sense. We're going to do a few exercises linking functions to your scripts and show you how to make more. For now, just keep thinking “mini-script” when I say “function” and keep playing with them.

---

## What You Should See

If you run `ex18.py`, you should see:

```
1  arg1: Zed, arg2: Shaw
2  arg1: Zed, arg2: Shaw
3  arg1: First!
4  I got nothin'.
```

Right away you can see how a function works. Notice that you used your functions the way you use things like `exists`, `open`, and other “commands.” In fact, I've been tricking you because in Python those “commands” are just functions. This means you can make your own commands and use them in your scripts too.

## Study Drills

Create a *function checklist* for later exercises. Write these checks on an index card and keep it by you while you complete the rest of these exercises

or until you feel you do not need the index card anymore:

1. Did you start your function definition with `def`?
2. Does your function name have only characters and `_` (underscore) characters?
3. Did you put `(` (an open parenthesis) right after the function name?
4. Did you put your arguments after `(` (the open parenthesis) separated by commas?
5. Did you make each argument unique (meaning no duplicated names)?
6. Did you put `):` (a close parenthesis and a colon) after the arguments?
7. Did you indent all lines of code you want in the function four spaces? No more, no less.
8. Did you “end” your function by going back to writing with no indent (“dedenting” we call it)?


When you run (“use” or “call”) a function, check these things:

1. Did you call/use/run this function by typing its name?
2. Did you put the `(` character after the name to run it?
3. Did you put the values you want into the parentheses separated by commas?
4. Did you end the function call with a `)` character?

Use these two checklists on the remaining lessons until you do not need them anymore.

Finally, repeat this a few times to yourself:

```
1  "To 'run,' 'call,' or 'use' a function all mean the same thing"
```



## Common Student Questions



**What's allowed for a function name?** The same as variable names. Anything that doesn't start with a number and is letters, numbers, and underscores will work.

**What does the `* in *args` do?** That tells Python to take all the arguments to the function and then put them in `args` as a list. It's like `argv` that you've been using but for functions. It's not normally used too often unless specifically needed.

**This feels really boring and monotonous.** That's good. It means you're starting to get better at typing in the code and understanding what it does. To make it less boring, take everything I tell you to type in, and then break it on purpose.

## Exercise 19. Functions and Variables

You're now going to combine functions with what you know of variables from previous exercises. As you know, a variable gives a piece of data a name so you can use it in your program. If you have this code:

```
1 x = 10
```

then you created a piece of data named `x` that is equal to the number 10.

You also know that you can call functions with parameters like this:

```
1 def print_one(arg1):  
2     print(f"arg1: {arg1}")
```

The parameter `arg1` is a variable similar to the `x` before, except it's created for you when you *call* the function like this:

```
1 print_one("First!")
```

In [Exercise 18](#) you learned how Python runs functions when you call them, but what happens if you did this:

```
1 y = "First!"  
2 print_one(y)
```

Instead of calling `print_one` directly with `"First!"` you're assigning `"First!"` to `y` and *then* passing `y` to `print_one`. Does this work? Here's a small sample code you can use to test this out in Jupyter:

```
1 def print_one(arg1):  
2     print(f"arg1: {arg1}")  
3  
4 y = "First!"  
5 print_one(y)
```

This shows how you can combine the concept of variables `y = "First!"` with calling functions that use the variables. Study this and try your own variations before working on this longer exercise, but first a bit of advice:

1. This one is long, so if you find it difficult to manage in Jupyter, then try typing it into an `ex19.py` file to run in Terminal.
2. As usual, you should type only a few lines at a time, but you'll have problems if you type only the first line of a function. You can solve this by using the `pass` keyword like this: `def some_func(some_arg): pass`. The `pass` word is how you make an empty function without causing an error.
3. If you want to see what each function is doing, you can use “debug printing” like this: `print(">>> I'm here", something)`. That will print out a message to help you “trace” through the code and see what something is in each function.

Listing 19.1: `ex19.py`

```
1  def cheese_and_crackers(cheese_count, boxes_of_crackers)
2      print(f"You have {cheese_count} cheeses!")
3      print(f"You have {boxes_of_crackers} boxes of crackers")
4      print("Man that's enough for a party!")
5      print("Get a blanket.\n")
6
7
8  print("We can just give the function numbers directly:")
9  cheese_and_crackers(20, 30)
10
11
12 print("OR, we can use variables from our script:")
13 amount_of_cheese = 10
14 amount_of_crackers = 50
15
16 cheese_and_crackers(amount_of_cheese, amount_of_crackers)
17
18
19 print("We can even do math inside too!")
```

```
19 print( "We can even do math inside too. " )
20 cheese_and_crackers(10 + 20, 5 + 6)
21
22
23 print("And we can combine the two, variables and math:")
24 cheese_and_crackers(amount_of_cheese + 100, amount_of_crackers + 1000)
```

## What You Should See

```
1 We can just give the function numbers directly:
2 You have 20 cheeses!
3 You have 30 boxes of crackers!
4 Man that's enough for a party!
5 Get a blanket.
6
7 OR, we can use variables from our script:
8 You have 10 cheeses!
9 You have 50 boxes of crackers!
10 Man that's enough for a party!
11 Get a blanket.
12
13 We can even do math inside too:
14 You have 30 cheeses!
15 You have 11 boxes of crackers!
16 Man that's enough for a party!
17 Get a blanket.
18
19 And we can combine the two, variables and math:
20 You have 110 cheeses!
21 You have 1050 boxes of crackers!
22 Man that's enough for a party!
23 Get a blanket.
```

## Study Drills

1. Did you remember to type only a few lines at a time? Did you use `pass` to make an empty function before filling it? If not, delete your code and do it again.
2. Change the name of `cheese_and_crackers` to have a spelling mistake and view the error message. Now fix it.
3. Delete one of the `+` symbols in the math to see what error you get.
4. Make changes to the math and then try to predict what output you'll get.
5. Change the variables and try to guess the output with those changes.

## Common Student Questions

This exercise has no questions yet, but you can ask me at [help@learncodethehardway.org](mailto:help@learncodethehardway.org) to get help. Maybe your question will show up here.

## Exercise 20. Functions and Files

Remember your checklist for functions, and then do this exercise paying close attention to how functions and files can work together to make useful stuff. You should also continue to type only a few lines before running your code. If you catch yourself typing too many lines, then delete them and do it again. Doing this uses python to train your understanding of Python.

Here's the code for this exercise. Once again, it's long, so if you find Jupyter is difficult to use, then write a `ex20.py` file and run it that way.

Listing 20.1: `ex20.py`

```
1  from sys import argv
2
3  script, input_file = argv
4
5  def print_all(f):
6      print(f.read())
7
8  def rewind(f):
9      f.seek(0)
10
11 def print_a_line(line_count, f):
12     print(line_count, f.readline())
13
14 current_file = open(input_file)
15
16 print("First let's print the whole file:\n")
17
18 print_all(current_file)
19
20 print("Now let's rewind, kind of like a tape.")
21
22 rewind(current_file)
```

```
23
24 print("Let's print three lines:")
25
26 current_line = 1
27 print_a_line(current_line, current_file)
28
29 current_line = current_line + 1
30 print_a_line(current_line, current_file)
31
32 current_line = current_line + 1
33 print_a_line(current_line, current_file)
```

Pay close attention to how we pass in the current line number each time we run `print_a_line`. There's nothing new in this exercise. It has functions, and you know those. It has files and you know those too. Just take your time with it and you'll get it.

## What You Should See

```
1 First let's print the whole file:
2
3 This is line 1
4 This is line 2
5 This is line 3
6
7 Now let's rewind, kind of like a tape.
8 Let's print three lines:
9 1 This is line 1
10
11 2 This is line 2
12
13 3 This is line 3
```

## Study Drills

1. Write English comments for each line to understand what that line does.
2. Each time `print_a_line` is run, you are passing in a variable `current_line`. Write out what `current_line` is equal to on each function call, and trace how it becomes `line_count` in `print_a_line`.
3. Find each place a function is used, and check its `def` to make sure that you are giving it the right arguments.
4. Research online what the `seek` function for `file` does. Try `pydoc file`, and see if you can figure it out from there. Then try `pydoc file.seek` to see what `seek` does.
5. Research the shorthand notation `+=`, and rewrite the script to use `+=` instead.

## Common Student Questions

**What is `f` in the `print_all` and other functions?** The `f` is a variable just like you had in other functions in [Exercise 18](#), except this time it's a file. A file in Python is kind of like an old tape drive on a mainframe or maybe a DVD player. It has a "read head," and you can "seek" this read head around the file to a position and then work with it there. Each time you do `f.seek(0)` you're moving to the start of the file. Each time you do `f.readline()` you're reading a line from the file and moving the read head to the right after the `\n` that ends that line. This will be explained more as you go on.

**Why does `seek(0)` not set the `current_line` to 0?** First, the `seek()` function is dealing in *bytes*, not lines. The code `seek(0)` moves the file to the 0 byte (first byte) in the file. Second, `current_line` is just a variable and has no real connection to the file at all. We are manually incrementing it.

**What is `+=`?** You know how in English I can rewrite "it is" as "it's"? Or I can rewrite "you are" as "you're"? In English this is called a



“contraction,” and this is kind of like a contraction for the two operations `=` and `+`. That means `x = x + y` is the same as `x += y`.

**How does `readline()` know where each line is?** Inside `readline()` is code that scans each byte of the file until it finds a `\n` character and then stops reading the file to return what it found so far. The file `f` is responsible for maintaining the current position in the file after each `readline()` call so that it will keep reading each line.

**Why are there empty lines between the lines in the file?** The `readline()` function returns the `\n` that’s in the file at the end of that line. Add an `end = ""` at the end of your `print` function calls to avoid adding double `\n` to every line.

## Exercise 21. Functions Can Return Something

You have been using the = character to name variables and set them to numbers or strings. We're now going to blow your mind again by showing you how to use = and a new Python word return to set variables to be a *value from a function*. There will be one thing to pay close attention to, but first type this in:

Listing 21.1: ex21.py

```
1  def add(a, b):
2      print(f"ADDING {a} + {b}")
3      return a + b
4
5  def subtract(a, b):
6      print(f"SUBTRACTING {a} - {b}")
7      return a - b
8
9  def multiply(a, b):
10     print(f"MULTIPLYING {a} * {b}")
11     return a * b
12
13  def divide(a, b):
14     print(f"DIVIDING {a} / {b}")
15     return a / b
16
17
18  print("Let's do some math with just functions!")
19
20  age = add(30, 5)
21  height = subtract(78, 4)
22  weight = multiply(90, 2)
23  iq = divide(100, 2)
24
25  print(f"Age: {age} Height: {height} Weight: {weight} IQ: {iq}")
```

```

25 print('Age: {age}, height: {height}, weight: {weight},
26
27
28 # A puzzle for the extra credit, type it in anyway.
29 print("Here is a puzzle.")
30
31 what = add(age, subtract(height, multiply(weight, divide
32
33 print("That becomes: ", what, "Can you do it by hand?")

```

We are now doing our own math functions for add, subtract, multiply, and divide. The important thing to notice is the last line where we say return a + b (in add). What this does is the following:

1. Our function is called with two arguments: a and b.
2. We print out what our function is doing, in this case “ADDING.”
3. Then we tell Python to do something kind of backward: we return the addition of a + b. You might say this as, “I add a and b and then return them.”
4. Python adds the two numbers. Then when the function ends, any line that runs it will be able to assign this a + b result to a variable.

As with many other things in this book, you should take this really slowly, break it down, and try to trace what’s going on. To help there is extra credit to solve a puzzle and learn something cool.

## What You Should See

```

1  Let's do some math with just functions!
2  ADDING 30 + 5
3  SUBTRACTING 78 - 4
4  MULTIPLYING 90 * 2
5  DIVIDING 100 / 2
6  Age: 35, Height: 74, Weight: 180, IQ: 50.0
7  Here is a puzzle.

```

```
8  DIVIDING 50.0 / 2
9  MULTIPLYING 180 * 25.0
10 SUBTRACTING 74 - 4500.0
11 ADDING 35 + -4426.0
12 That becomes: -4391.0 Can you do it by hand?
```

## Study Drills

1. If you aren't really sure what return does, try writing a few of your own functions and have them return some values. You can return anything that you can put to the right of an `=`.
2. At the end of the script is a puzzle. I'm taking the return value of one function and *using* it as the argument of another function. I'm doing this in a chain so that I'm kind of creating a formula using the functions. It looks really weird, but if you run the script, you can see the results. What you should do is try to figure out the normal formula that would re-create this same set of operations.
3. Once you have the formula worked out for the puzzle, get in there and see what happens when you modify the parts of the functions. Try to change it on purpose to make another value.
4. Do the inverse. Write a simple formula and use the functions in the same way to calculate it.

This exercise might really wreck your brain, but take it slow and and treat it like a little game. Figuring out puzzles like this is what makes programming fun, so I'll be giving you more little problems like this as we go.

## Common Student Questions

### **Why does Python print the formula or the functions “backward”?**

It's not really backward, it's “inside out.” You'll see how it works when you start breaking down the function into separate formulas and functions. Try to understand what I mean by “inside out” rather than “backward.”

**How can I use `input()` to enter my own values?** Remember `int(input())`? The problem with that is then you can't enter floating point, so also try using `float(input())` instead.

**What do you mean by “write out a formula”?** Try  $24 + 34 / 100 - 1023$  as a start. Convert that to use the functions. Now come up with your own similar math equation, and use variables so it's more like a formula.

## Exercise 22. Strings, Bytes, and Character Encodings

To do this exercise you'll need to *download* a text file that I've written named `languages.txt`. This file was created with a list of human languages to demonstrate a few interesting concepts:

- How modern computers store human languages for display and processing and how Python 3 calls these strings
- How you must “encode” and “decode” Python's strings into a type called bytes
- How to handle errors in your string and byte handling
- How to read code and find out what it means even if you've never seen it before

You can get this file by doing a right click with your mouse and selecting “Download” to download the file reliably. Use the link <https://learnpythonthehardway.org/python3/languages.txt> to download the file.

In addition, you'll get a brief glimpse of the Python 3 `if`-statement and `lists` for processing a list of things. You don't have to master this code or understand these concepts right away. You'll get plenty of practice in later exercises. For now your job is to get a taste of the future and learn the four topics in the preceding list.

---

### Warning!

This exercise is hard! There's a lot of information in it that you need to understand, and it's information that goes deep into computers. This exercise is complex because Python's strings are complex and difficult to use. I recommend you take this exercise painfully slow.

Write down every word you don't understand, and look it up or research it. Take a paragraph at a time if you must. You can continue with other exercises while you study this one, so don't get stuck here. Just chip away at it for as long as it takes.

---

## Initial Research

You will create a file named `ex22.py` and run it in the shell for this exercise. Be sure you know how to do that, and if not, revisit [Exercise 0](#) where you learned how to run Python code from the Terminal.

I'm going to teach you how to research a piece of code to expose its secrets. You'll need the `languages.txt` file for this code to work, so make sure you download it first. The `languages.txt` file simply contains a list of human language names that are encoded in UTF-8.

Listing 22.1: `ex22.py`

```
1  import sys
2  script, input_encoding, error = sys.argv
3
4
5  def main(language_file, encoding, errors):
6      line = language_file.readline()
7
8      if line:
9          print_line(line, encoding, errors)
10         return main(language_file, encoding, errors)
11
12
13 def print_line(line, encoding, errors):
14     next_lang = line.strip()
15     raw_bytes = next_lang.encode(encoding, errors=errors)
16     cooked_string = raw_bytes.decode(encoding, errors=errors)
17
18     print(raw_bytes, "<===>", cooked_string)
19
20
```

```
19
20
21 languages = open("languages.txt", encoding="utf-8")
22
23 main(languages, input_encoding, error)
```

Try to study this code by writing down each thing you don't recognize, and then search for it with the usual python `THING` site:python.org. For example, if you don't know what `encode()` does, then search for `python encode` site:python.org. Once you've read the documentation for everything you don't know, continue with this exercise.

Once you have that you'll want to run this Python script in your shell to play with it. Here are some example commands I used to test it:

```
1 python ex22.py "utf-8" "strict"
2 python ex22.py "utf-8" "ignore"
3 python ex22.py "utf-8" "replace"
```

See the documentation for the `str.encode()` function for more options.

---

## Warning!

You'll notice I'm using images here to show you what you should see. After extensive testing it turns out that so many people have their computers configured to not display utf-8 that I had to use images so you'll know what to expect. Even my own typesetting system (LaTeX) couldn't handle these encodings, forcing me to use images instead. If you don't see this, then your terminal is most likely not able to display utf-8, and you should try to fix that.

---

These examples use the `utf-8`, `utf-16`, and `big5` encodings to demonstrate the conversion and the types of errors you can get. Each of these names are called a “codec” in Python 3, but you use the parameter “encoding”. At the end of this exercise there's a list of the available encodings if you want to



try more. I'll cover what all of this output means shortly. You're only trying to get an idea of how this works so we can talk about it.

After you've run it a few times, go through your list of symbols and make a guess as to what they do. When you've written down your guesses, try looking the symbols up online to see if you can confirm your hypothesis. Don't worry if you have no idea how to search for them. Just give it a try.

## Switches, Conventions, and Encodings

Before I can get into what this code means, you need to learn some basics about how data is stored in a computer. Modern computers are incredibly complex, but at their core they are like a huge array of light switches. Computers use electricity to flip switches on or off. These switches can represent 1 for on, and 0 for off. In the old days there were all kinds of weird computers that did more than just 1 or 0, but these days it's just 1s and 0s. One represents energy, electricity, on, power, substance. Zero represents off, done, gone, power down, the lack of energy. We call these 1s and 0s "bits."

Now, a computer that lets you work only with 1 and 0 would be both horribly inefficient and incredibly annoying. Computers take these 1s and 0s and use them to encode larger numbers. At the small end a computer will use 8 of these 1s and 0s to encode 256 numbers (0–255). What does "encode" mean though? It's nothing more than an agreed upon standard for how a sequence of bits should represent a number. It's a convention humans picked or stumbled on that says that 00000000 would be 0, 11111111 would be 255, and 00001111 would be 15. There were even huge wars in the early history of computers on nothing more than the order of these bits because they were simply conventions we all had to agree on.

Today we call a "byte" a sequence of 8 bits (1s and 0s). In the old days everyone had their own convention for a byte, so you'll still run into people who think that this term should be flexible and handle sequences of 9 bits, 7 bits, 6 bits, but now we just say it's 8 bits. That's our convention, and that convention defines our encoding for a byte. There are further conventions for encoding large numbers using 16, 32, 64, and even more bits if you get

into really big math. There are entire standards groups who do nothing but argue about these conventions and then implement them as encodings that eventually turn switches on and off.

Once you have bytes, you can start to store and display text by deciding on another convention for how a number maps to a letter. In the early days of computing there were many conventions that mapped 8 or 7 bits (or less or more) onto lists of characters kept inside a computer. The most popular convention ended up being American Standard Code for Information Interchange, or ASCII. This standard maps a number to a letter. The number 90 is Z, which in bits is 1011010, which gets mapped to the ASCII table inside the computer.

You can try this out in Python right now:

```
1  >>> 0b1011010
2  90
3  >>> ord('Z')
4  90
5  >>> chr(90)
6  'Z'
7  >>>
```

First, I write the number 90 in binary, then I get the number based on the letter Z, then I convert the number to the letter Z. Don't worry about needing to remember this though. I think I've had to do it twice the entire time I've used Python.

Once we have the ASCII convention for encoding a character using 8 bits (a byte), we can then “string” them together to make a word. If I want to write my name “Zed A. Shaw,” I just use a sequence of bytes that is [90, 101, 100, 32, 65, 46, 32, 83, 104, 97, 119]. Most of the early text in computers was nothing more than sequences of bytes, stored in memory, that a computer used to display text to a person. Again, this is just a sequence of conventions that turned switches on and off.

The problem with ASCII is that it only encodes English and maybe a few other similar languages. Remember that a byte can hold 256 numbers (0–255, or 00000000–11111111). Turns out, there are *lots* more characters than

256 used throughout the world's languages. Different countries created their own encoding conventions for their languages, and that mostly worked, but many encodings could handle only one language. That meant if you want to put the title of an American English book in the middle of a Thai sentence, you were kind of in trouble. You'd need one encoding for Thai and another for English.

To solve this problem a group of people created Unicode. It sounds like "encode," and it is meant to be a "universal encoding" of all human languages. The solution Unicode provides is like the ASCII table, but it's huge by comparison. You can use 32 bits to encode a Unicode character, and that is more characters than we could possibly find. A 32-bit number means we can store 4,294,967,295 characters ( $2^{32}$ ), which is enough space for every possible human language and probably a lot of alien ones too. Right now we use the extra space for important things like poop and smile emojis.

We now have a convention for encoding any character we want, but 32 bits is 4 bytes ( $32/8 == 4$ ), which means there is so much wasted space in most text we want to encode. We can also use 16 bits (2 bytes), but still there's going to be wasted space in most text. The solution is to use a clever convention to encode most common characters using 8 bits and then "escape" into larger numbers when we need to encode more characters. That means we have one more convention that is nothing more than a compression encoding, making it possible for most common characters to use 8 bits and then escape out into 16 or 32 bits as needed.

The convention for encoding text in Python is called "utf-8", which means "Unicode Transformation Format 8 Bits." It is a convention for encoding Unicode characters into sequences of bytes (which are sequences of bits (which turn sequences of switches on and off)). You can also use other conventions (encodings), but utf-8 is the current standard.

## Dissecting the Output

We can now look at the output of the previous commands. Let's take just that first command and the first few lines of output:

```

$ python ex22.py "utf-8" "strict"
b'Afrikaans' <====> Afrikaans
b'\xe1\x8a\xa0\xe1\x88\x9b\xe1\x88\xad\xe1\x8a\x9b' <====> አማርኛ
b'\xd0\x90\xd2\xa7\xd1\x81\xd1\x88\xd3\x99\xd0\xb0' <====> Анҗсшәә
b'\xd8\xa7\xd9\x84\xd8\xb9\xd8\xb1\xd8\xa8\xd9\x8a\xd8\xa9' <====> العربية
b'Aragon\xc3\xa9s' <====> Aragonés
b'Arpetan' <====> Arpetan
b'Az\xc9\x99rbaycanca' <====> Azərbaycanca
b'Bamanankan' <====> Bamanankan
b'\xe0\xa6\xac\xe0\xa6\xbe\xe0\xa6\x82\xe0\xa6\xb2\xe0\xa6\xbe' <====> বাংলা
b'B\xc3\xa2n-l\xc3\xaam-g\xc3\xba' <====> Bân-lâm-gú
b'\xd0\x91\xd0\xb5\xd0\xbb\xd0\xb0\xd1\x80\xd1\x83\xd1\x81\xd0\xba\xd0\xb0\xd1\x8f' <====> Беларуская
b'\xd0\x91\xd1\x8a\xd0\xbb\xd0\xb3\xd0\xb0\xd1\x80\xd1\x81\xd0\xba\xd0\xb8' <====> Български
b'Boarisch' <====> Boarisch
b'Bosanski' <====> Bosanski
b'\xd0\x91\xd1\x83\xd1\x80\xd1\x8f\xd0\xb0\xd0\xb4' <====> Буряад

```

The `ex22.py` script is taking bytes written inside the `b''` (byte string) and converting them to the UTF-8 (or other) encoding you specified. On the left are the numbers for each byte of the utf-8 (shown in hexadecimal), and the right has the character output as actual utf-8. The way to think of this is on the left side of `<====>` are the Python numerical bytes, or the “raw” bytes Python uses to store the string. You specify this with `b''` to tell Python this is “bytes”. These raw bytes are then displayed “cooked” on the right so you can see the real characters in your terminal.

## Dissecting the Code

We have an understanding of strings and byte sequences. In Python a string is a UTF-8 encoded sequence of characters for displaying or working with text. The bytes are then the “raw” sequence of bytes that Python uses to store this UTF-8 string and start with a `b'` to tell Python you are working with raw bytes. This is all based on conventions for how Python wants to work with text. Here’s a Python session showing me encoding strings and decoding bytes:

```
[2]: raw_bytes = b'\xe6\x96\x87\xe8\xa8\x80'  
raw_bytes.decode()
```

```
[2]: '文言'
```

```
[3]: utf_string = '文言'  
utf_string.encode()
```

```
[3]: b'\xe6\x96\x87\xe8\xa8\x80'
```

```
[4]: raw_bytes == utf_string.encode()
```

```
[4]: True
```

```
[6]: utf_string == raw_bytes.decode()
```

```
[6]: True
```

All you need to remember is if you have raw bytes, then you must use `.decode()` to get the string. Raw bytes have no convention to them. They are just sequences of bytes with no meaning other than numbers, so you must tell Python to “decode this into a utf string.” If you have a string and want to send it, store it, share it, or do some other operation, then usually it’ll work, but sometimes Python will throw up an error saying it doesn’t know how to “encode” it. Again, Python knows its internal convention, but it has no idea what convention you need. In that case, you must use `.encode()` to get the bytes you need.

The way to remember this (even though I look it up almost every time) is to remember the mnemonic “DBES,” which stands for “Decode Bytes Encode Strings.” I say “dee bess” in my head when I have to convert bytes and strings. When you have bytes and need a string, “Decode Bytes.” When you have a string and need bytes, “Encode Strings.”

With that in mind, let’s break down the code in `ex22.py` line by line:

**1-2** I start with your usual command line argument handling that you already know.

**5** I start the main meat of this code in a function conveniently called `main`. This will be called at the end of this script to get things going.

**6** The first thing this function does is read one line from the `languages` file it is given. You have done this before, so there’s

nothing new here. Just `readline` as before when dealing with text files.

**8** Now I use something *new*. You will learn about this in the second half of the book, so consider this a teaser of interesting things to come. This is an `if`-statement, and it lets you make decisions in your Python code. You can “test” the truth of a variable and, based on that truth, run a piece of code or not run it. In this case I’m testing whether `line` has something in it. The `readline` function will return an empty string when it reaches the end of the file and `if line` simply tests for this empty string. As long as `readline` gives us something, this will be true, and the code *under* (indented in, lines 9–10) will run. When this is false, Python will skip lines 9–10.

**9** I then call a separate function to do the actual printing of this line. This simplifies my code and makes it easier for me to understand it. If I want to learn what this function does, I can jump to it and study. Once I know what `print_line` does, I can attach my memory to the name `print_line` and forget about the details.

**10** I have written a tiny yet powerful piece of magic here. I am calling `main` again inside `main`. Actually, it’s not magic since nothing really is magical in programming. All the information you need is there. This looks like I am calling the function *inside* itself, which seems like it should be illegal to do. Ask yourself, why should that be illegal? There’s no technical reason why I can’t call any function I want right there, even this `main` function. If a function is simply a jump to the top where I’ve named it `main`, then calling this function at the end of itself would ... jump back to the top and run it again. That would make it loop. Now look back at line 8, and you’ll see the `if`-statement keeps this function from looping forever. Carefully study this because it is a significant concept, but don’t worry if you can’t grasp it right away.

**13** I now start the definition for the `print_line` function, which does the actual encoding of each line from the `languages.txt` file.

- 14** This is a simple stripping of the trailing `\n` on the `line` string.
- 15** Now I *finally* take the language I've received from the `languages.txt` file and "encode" it into the raw bytes. Remember the "DBES" mnemonic. "Decode Bytes, Encode Strings." The `next_lang` variable is a string, so to get the raw bytes I must call `.encode()` on it to "Encode Strings." I pass to `encode()` the encoding I want and how to handle errors.
- 16** I then do the extra step of showing the inverse of line 15 by creating a `cooked_string` variable from the `raw_bytes`. Remember, "DBES" says I "Decode Bytes," and `raw_bytes` is bytes, so I call `.decode()` on it to get a Python string. This string should be the same as the `next_lang` variable.
- 18** Then I simply print them both out to show you what they look like.
- 21** I'm done defining functions, so now I want to open the `languages.txt` file.
- 23** The end of the script simply runs the `main` function with all the correct parameters to get everything going and kick-start the loop. Remember that this then jumps to where the `main` function is defined on line 5, and on line 10 `main` is called again, causing this to keep looping. The `if line:` on line 8 will prevent our loop from going forever.

## Encodings Deep Dive

We can now use our little script to explore other encodings. Here's me playing with different encodings and seeing how to break them:

```

$ python ex22.py "utf-16" "strict"
b'\xff\xfeA\x00f\x00r\x00i\x00k\x00a\x00a\x00n\x00s\x00' <==> Afrikaans
b'\xff\xfe\xa0\x12\x1b\x12-\x12\x9b\x12' <==> አፋሪኛ
b'\xff\xfe\x10\x04\xa7\x04A\x04H\x04\xd9\x040\x04' <==> АҝCШәә
b'\xff\xfe'\x06D\x069\x061\x06(\x06J\x06)\x06' <==> دېرغول
...

$ python ex22.py "big5" "strict"
b'Afrikaans' <==> Afrikaans
Traceback (most recent call last):
  # cut the stack trace
UnicodeEncodeError: 'big5' codec can't encode character '\u12a0' in position 0: illegal multibyte sequence
$

```

First, I’m doing a simple UTF-16 encoding so you can see how it changes compared to UTF-8. You can also use “utf-32” to see how that’s even bigger and get an idea of the space saved with UTF-8. After that I try Big5, and you’ll see that Python does *not* like that at all. It throws up an error that “big5” can’t encode some of the characters at position 0 (which is super helpful). One solution is to tell Python to “replace” any bad characters for the Big5 encoding. Try that and you’ll see it puts a ? character wherever it finds a character that doesn’t match the Big5 encoding system.

## Breaking It

Rough ideas include the following:

1. Find strings of text encoded in other encodings and place them in the `ex22.py` file to see how it breaks.
2. Find out what happens when you give an encoding that doesn’t exist.
3. Extra challenging: Rewrite this using the `b' '` bytes instead of the UTF-8 strings, effectively reversing the script.
4. If you can do that, then you can also *break* these bytes by removing some to see what happens. How much do you need to remove to cause Python to break? How much can you remove to damage the string output but pass Python’s decoding system?



5. Use what you learned from #4 to see if you can mangle the files.  
What errors do you get? How much damage can you cause and get the file past Python's decoding system?

## Exercise 23. Introductory Lists

Most programming languages have some way to store data inside the computer. Some languages only have raw memory locations, but programmers easily make mistakes when that's the case. In modern languages you're provided with some core ways to store data called "data structures." A data structure takes pieces of data (integers, strings, and even other data structures) and organizes them in some useful way. In this exercise we'll learn about the sequence style of data structures called a "list" or "Array" depending on the language.

Python's simplest sequence data structure is the `list`, which is an ordered list of things. You can access the elements of a `list` randomly, in order, extend it, shrink it, and do most anything else you could do to a sequence of things in real life.

You make a `list` like this:

```
1 fruit = ["apples", "oranges", "grapes"];
```

That's all. Just put `[` (left-square-bracket) and `]` (right-square-bracket) around the `list` of things and separate them with commas. You can also put anything you want into a `list`, even other `lists`:

```
1 inventory = [ ["Buick", 10], ["Corvette", 1], ["Toyota", 4]];
```

In this code I have a `list`, and that `list` has three `lists` inside it. Each of those `lists` then has a name of a car type and the count of inventory. Study this and make sure you can take it apart when you read it. Storing `lists` inside `lists` inside other data structures is very common.

## Accessing Elements of a List

What if you want the first element of the `inventory` list? How about the number of Buick cars you have on inventory? You do this:

```
1  # get the buick record
2  buicks = inventory[0]
3  buick_count = buicks[1]
4  # or in one move
5  count_of_buicks = inventory[0][1]
```

In the first two lines of code (after the comment) I do a two-step process. I use the `inventory[0]` code to get the *first* element. If you're not familiar with programming languages, most start at 0, not 1, as that makes math work better in most situations. The use of `[]` right after a variable name tells Python that this is a "container thing" and says we want to "index into this thing with this value," in this case 0. In the next line I take the `buicks[1]` element and get the count 10 from it.

You don't have to do that though as you can chain the uses of `[]` in a sequence so that you dive deeper into a list as you go. In the last line of code I do that with `inventory[0][1]`, which says "get the 0 element and then get the 1 element of *that*."

Here's where you're going to make a mistake. The second `[1]` does not mean to get the entire `["Buick", [rcurvearrowse] 10]`. It's not linear, it's "recursive," meaning it dives into the structure. You are getting 10 in `["Buick", [rcurvearrowse] 10]`. It is more accurately just a combination of the first two lines of code.

## Practicing Lists

Lists are simple enough, but you need practice accessing different parts of very complicated lists. It's important that you can *correctly* understand how an index into a nested list will work. The best way to do that is to drill using such a list in Jupyter.

How this works is I have a series of lists in the following code. You are to type this code in like normal, and then you have to use Python to access the

elements so you get the same answers as I do.

## The Code

To complete this challenge you need this code:

Listing 23.1: ex23.py

```
1  fruit = [  
2      ['Apples', 12, 'AAA'], ['Oranges', 1, 'B'],  
3      ['Pears', 2, 'A'], ['Grapes', 14, 'UR']]  
4  
5  cars = [  
6      ['Cadillac', ['Black', 'Big', 34500]],  
7      ['Corvette', ['Red', 'Little', 1000000]],  
8      ['Ford', ['Blue', 'Medium', 1234]],  
9      ['BMW', ['White', 'Baby', 7890]]  
10 ]  
11  
12 languages = [  
13     ['Python', ['Slow', ['Terrible', 'Mush']]],  
14     ['JavaScript', ['Moderate', ['Alright', 'Bizarre']]],  
15     ['Perl6', ['Moderate', ['Fun', 'Weird']]],  
16     ['C', ['Fast', ['Annoying', 'Dangerous']]],  
17     ['Forth', ['Fast', ['Fun', 'Difficult']]],  
18 ]
```

It's fine to copy-paste this code since the point of this exercise is learning how to access data, but if you want extra practice typing Python, then enter it in manually.

## The Challenge

I will give you a list name and a piece of data in the list. Your job is to figure out what indexes you need to get that information. For example, if I tell you `fruit 'AAA'`, then your answer is `fruit[0][2]`. You should attempt

to do this in your head by looking at the code and then test your guess in the Jupyter.

## **fruit challenge**

You need to get all of these elements out of the `fruit` variable:

- 12
- 'AAA'
- 2
- 'Oranges'
- 'Grapes'
- 14
- 'Apples'

## **cars challenge**

You need to get all of these elements out of the `cars` variable:

- 'Big'
- 'Red'
- 1234
- 'White'
- 7890
- 'Black'
- 34500
- 'Blue'

## **languages challenge**

You need to get all of these elements out of the `languages` variable:

- ‘Slow’
- ‘Alright’
- ‘Dangerous’
- ‘Fast’
- ‘Difficult’
- ‘Fun’
- ‘Annoying’
- ‘Weird’
- ‘Moderate’

## Final Challenge

You now have to figure out what this code spells out:

```
1 cars[1][1][1]
2 cars[1][1][0]
3 cars[1][0]
4 cars[3][1][1]
5 fruit[3][2]
6 languages[0][1][1][1]
7 fruit[2][1]
8 languages[3][1][0]
```

Don’t attempt to run this in Jupyter first. Instead, try to work out manually what each line will spell out, and then test it in Jupyter.

## Exercise 24. Introductory Dictionaries

In this exercise we'll use the same data from the previous exercise on lists and use it to learn about Dictionaries or dicts.

### Key/Value Structures

You use key=value data all the time without realizing it. When you read an email, you might have:

```
1 From: j.smith@example.com
2 To: zed.shaw@example.com
3 Subject: I HAVE AN AMAZING INVESTMENT FOR YOU!!!
```

On the left are the keys (From, To, Subject) which are *mapped* to the contents on the right of the `:`. Programmers say the key is “mapped” to the value, but they could also say “set to” as in, “I set From to `j.smith@example.com`.” In Python I might write this same email using a data object like this:

```
1 email = {
2     "From": "j.smith@example.com",
3     "To": "zed.shaw@example.com",
4     "Subject": "I HAVE AN AMAZING INVESTMENT FOR YOU!!!"
5 }
```

You create a data object by:

1. Opening it with a `{` (curly-brace)
2. Writing the key, which is a string here, but can be numbers, or almost anything
3. Writing a `:` (colon)
4. Writing the value, which can be anything that's valid in Python

Once you do that, you can access this Python email like this:

```
1 email["From"]
2 'j.smith@example.com'
3
4 email["To"]
5 'zed.shaw@example.com'
6
7 email["Subject"]
8 'I HAVE AN AMAZING INVESTMENT FOR YOU!!!'
```

You'll notice that this is very similar to how you access variables and functions in a module that you require. Using the `.` (dot) is a primary way you can access parts of many data structures in Python. You can also access this data using the `[]` syntax from the previous exercise:

```
1 email['To']
2 'zed.shaw@example.com'
3
4 email['From']
5 'j.smith@example.com'
```

The only difference from `list` indexes is that you use a string ('From') instead of an integer. However, you could use an integer as a key if you want (more on that soon).

## Combining Lists with Data Objects

A common theme in programming is combining components for surprising results. Sometimes the surprise is a crash or a bug. Other times the surprise is a novel new way to accomplish some task. Either way, what happens when you make novel combinations isn't really a surprise or a secret. To *you* it may be surprising, but there is usually an explanation somewhere in the language specification (even if that reason is absolutely stupid). There is no magic in your computer, just complexity you don't understand.

A good example of combining Python components is putting data Objects inside `lists`. You can do this:



```
1 messages = [  
2     {to: 'Sun', from: 'Moon', message: 'Hi!'},  
3     {to: 'Moon', from: 'Sun', message: 'What do you want Sun?'},  
4     {to: 'Sun', from: 'Moon', message: 'I'm awake!'},  
5     {to: 'Moon', from: 'Sun', message: 'I can see that Sun.'}  
6 ];
```

Once I do that I can now use list syntax to access the data objects like this:

```
1 messages[0].to  
2 'Sun'  
3  
4 messages[0].from  
5 'Moon'  
6 messages[0].message  
7 'Hi!'  
8  
9 messages[1]['to']  
10 'Moon'  
11  
12 messages[1]['from']  
13 'Sun'  
14  
15 messages[1]['message']  
16 'What do you want Sun?'
```

Notice how I can also use the . (dot) syntax on the data object right after doing `messages[0]`? Again, you can try combining features to see if they work, and if they do, go find out why because there's always a reason (even if it's stupid).

## The Code

You are now going to repeat the exercise you did with lists and write out three data objects I've crafted. Then you'll type them into Python and attempt to access the data I give you. Remember to try to do this in your

head and then check your work with Python. You should also practice doing this to list and dict structures until you're confident you can access the contents. You'll realize that the data is the same, it's simply been restructured.

Listing 24.1: ex24.py

```
1  fruit = [  
2      {kind: 'Apples', count: 12, rating: 'AAA'},  
3      {kind: 'Oranges', count: 1, rating: 'B'},  
4      {kind: 'Pears', count: 2, rating: 'A'},  
5      {kind: 'Grapes', count: 14, rating: 'UR'}  
6  ];  
7  
8  cars = [  
9      {type: 'Cadillac', color: 'Black', size: 'Big', miles:  
10     {type: 'Corvette', color: 'Red', size: 'Little', mi  
11     {type: 'Ford', color: 'Blue', size: 'Medium', miles  
12     {type: 'BMW', color: 'White', size: 'Baby', miles: '  
13  ];  
14  
15  languages = [  
16      {name: 'Python', speed: 'Slow', opinion: ['Terrible  
17      {name: 'JavaScript', speed: 'Moderate', opinion: ['  
18      {name: 'Perl6', speed: 'Moderate', opinion: ['Fun',  
19      {name: 'C', speed: 'Fast', opinion: ['Annoying', 'Da  
20      {name: 'Forth', speed: 'Fast', opinion: ['Fun', 'Di  
21  ];
```

## What You Should See

Keep in mind that you're doing some complicated data access moves here, so take it slow. You have to go through the data variable you assign the module to, and then access lists, followed by data objects, and in some cases another list.

# The Challenge

I will give you the exact same set of data elements for you to get. Your job is to figure out what indexing you need to get that information. For example, if I tell you fruit 'AAA', then your answer is fruit[0].rating. You should attempt to do this in your head by looking at the code and then test your guess in the python shell.

## fruit challenge

You need to get all of these elements out of the fruit variable:

- 12
- 'AAA'
- 2
- 'Oranges'
- 'Grapes'
- 14
- 'Apples'

## cars challenge

You need to get all of these elements out of the cars variable:

- 'Big'
- 'Red'
- 1234
- 'White'
- 7890
- 'Black'
- 34500
- 'Blue'

## languages challenge

You need to get all of these elements out of the `languages` variable:

- ‘Slow’
- ‘Alright’
- ‘Dangerous’
- ‘Fast’
- ‘Difficult’
- ‘Fun’
- ‘Annoying’
- ‘Weird’
- ‘Moderate’

## Final Challenge

Your final challenge is to write out the Python code that writes out the same song lyric from [Exercise 23](#). Again, take it slow and try to do it in your head before seeing whether you get it right. If you get it wrong, take the time to understand why you got it wrong. For comparison, I wrote out the lyrics in my head in one shot and didn’t get it wrong. I am also way more experienced than you are, so you will probably make some mistakes and that is alright.

You didn’t know those were song lyrics? It’s a Prince song called “Little Red Corvette.” You are now ordered to listen to 10 Prince songs before you continue with this book or we cannot be friends anymore. Anymore!

## Exercise 25. Dictionaries and Functions

In this exercise we're going to do something fun by combining functions with dicts. The purpose of this exercise is to confirm that you can combine different things in Python. Combination is a key aspect of programming, and you'll find that many "complex" concepts are nothing more than a combination of simpler concepts.

### Step 1: Function Names Are Variables

To prepare we first have to confirm that a function's name is just like other variables. Take a look at this code:

```
1  def print_number(x):  
2      print("NUMBER IS", x)  
3  
4  rename_print = print_number  
5  rename_print(100)  
6  print_number(100)
```

If you run this code, you'll see that `rename_print` does the exact same thing as `print_number`, and that's because they are the same. The name of a function is the same as a variable, and you can reassign the name to another variable. It's the same as doing this:

```
1  x = 10  
2  y = x
```

Play around with this until you get the idea. Make your own functions and then assign them to new names until you get that idea.

### Step 2: Dictionaries with Variables

It might be obvious, but just in case you haven't made the connection, you can put a variable into a dict:

```
1 color = "Red"
2
3 corvette = {
4     "color": color
5 }
6
7 print("LITTLE", corvette["color"], "CORVETTE")
```

This next piece of the puzzle makes sense, since you can put values into a dict like numbers and strings. You can also assign those same values to variables, so it makes sense you can combine both and put a variable into a dict.

### Step 3: Dictionaries with Functions

You should be seeing where this is going, but now we can combine these concepts to put a function in a dict:

```
1 def run():
2     print("VROOM")
3
4 corvette = {
5     "color": "Red",
6     "run": run
7 }
8
9 print("My", corvette["color"], "can go")
10 corvette["run"]()
```

I've taken the color variable from before and simply put it right in the dict for the corvette. Then I made a function run and put that into the corvette as well. The tricky part is that last line `corvette["run"]()`, but see if you can figure it out based on what you know. Take some time to write out a description of what this line is doing before continuing on.

## Step 4: Deciphering the Last Line

The trick to deciphering that last line `corvette["run"]()` is to separate out each piece of it. What confuses people about lines like this is they see one single thing, “run the corvette.” The truth is this line is composed of *many* things working together in combination. If we break this apart, we could have this code:

```
1  # get the run fuction out of the corvette dict
2  myrun = corvette["run"]
3  # run it
4  myrun()
```

Even those two lines isn't the entire story, but that shows you this is at *least* two operations on one line: get the function with `["run"]` and then run the function with `()`. To break this down further we can write:

1. `corvette` tells Python to load the dict
2. `[` tells Python to start an index into `corvette`
3. `"run"` tells Python to use `"run"` as the key to search the dict
4. `]` tells Python you are done and it should complete the index
5. Python then *returns* the contents of `corvette` that match the key `"run"`, which is the previous `run()` function
6. Python now has the `run` function, so `()` tells Python to call it like you would any other function

Take some time to understand how this is working, and write your own functions on the `corvette` to make it do more things.

## Study Drill

You now have a nice piece of code that's controlling a car. In this Study Drill you're going to create a new function that *creates any car*. Your *creator function* should meet these requirements:

1. It should take parameters to set things like the color, speed, or anything else your cars can do.
2. It should create a `dict` that has the correct settings and already contains all the functions you've created.
3. It should return this `dict` so people can assign the results to anything they want and use later.
4. It should be written so that someone can create any number of different cars and each one they make is independent.
5. Your code should test #4 by changing settings in a few different cars and then confirming they didn't change in other cars.

This challenge is different because I'll show you the answer to the challenge in a later exercise. If you struggle with this challenge, then shelve it for a bit and move on. You'll see this again shortly.



## Exercise 26. Dictionaries and Modules

In this exercise you're going to explore how the `dict` works with modules. You've been using modules any time you use `import` to add “features” to your own Python source. You did this the most in [Exercise 17](#), so it might be good to go review that exercise before you begin this one.

### Step 1: Review of `import`

The first step is review how `import` works and develop that knowledge further. Take some time to enter this code into a Python file named `ex26.py`. You can do this in Jupyter by creating a file (left side, blue `+` button) with that name:

Listing 26.1: `ex26.py`

```
1  name = "Zed"
2  height = 74
```

Once you've created this file you can import it with this:

Listing 26.2: `ex26_code.py`

```
1  import ex26
```

This will bring the contents of `ex26.py` into your Jupyter lab so you can access them like this:

Listing 26.3: `ex26_code.py`

```
1  print("name", ex26.name)
2  print("height", ex26.height)
```

Take some time to play with this as much as possible. Try adding new variables and doing the import again to see how that works.

## Step 2: Find the `__dict__`

Once you understand that the `import` is the contents of `ex26.py` to your lab, you can start investigating the `__dict__` variable like this:

Listing 26.4: `ex26_code.py`

```
1  from pprint import pprint
2
3  pprint(ex26.__dict__)
```

The `pprint` function is a “pretty printer” that will print the `__dict__` in a better format.

With `pprint` you suddenly see that `ex26` has a “hidden” variable called `__dict__`, which is *literally* a `dict` that contains everything in the module. You’ll find this `__dict__` and many other secret variables all over Python. The contents of `__dict__` contain quite a few things that aren’t your code, but that’s simply things Python needs to work with the module.

These variables are so hidden that even top professionals forget they exist. Many of these programmers believe that a module is *totally* different from a `dict` when internally a module uses a `__dict__`, which means it *is* the same as a `dict`. The only difference is Python has some syntax that lets you access a module using the `.` operator instead of the `dict` syntax, but you *can* still access the contents as a `dict`:

Listing 26.5: `ex26_code.py`

```
1  print("height is", ex26.height)
2  print("height is also", ex26.dict ['height'])
```

You'll get the same output for both syntaxes, but the `.` module syntax is definitely easier.

### Step 3: Change the `__dict__`

If a module is really a `dict` inside, then that means changing the contents of `__dict__` should also change the variables in the module. Let's try it:

Listing 26.6: `ex26_code.py`

```
1  print(f"I am currently {ex26.height} inches tall.")
2
3  ex26.__dict__['height'] = 1000
4  print(f"I am now {ex26.height} inches tall.")
5
6  ex26.height = 12
7  print(f"Oops, now I'm {ex26.__dict__['height']} inches tall.")
```

As you can see, the variable `ex26.height` changes when you change `ex26.__dict__['height']`, which proves that the module is really the `__dict__`.

This means that the `.` operator is being translated into a `__dict__[ ]` access operation. I want you to remember this for later, because many times when beginning programmers see `ex26.height`, they think this is a single unit of code. It is actually three or four separate operations:

1. Find `ex26`.
2. Find the `ex26.__dict__`.
3. Index into `__dict__` with `"height"`.
4. Return that value.

Once you make this connection you'll start to understand how the `.` works.

## Study Drill: Find the “Dunders”


The `__dict__` variables are typically called “double underscore” variables, but programmers are a lazy bunch so we just call them “dunder variables.” For this final step in learning about dunder variables you’ll visit the Python documentation for the data model, which describes how many of these dunders are used.

This is a large document, and its writing style is very dry, so the best way to study it is search for `__` (double underscore) and then find a way to access this variable based on its description. For example, you can try to access the `__doc__` on almost anything:

Listing 26.7: `ex26_code.py`

---

```
1  from pprint import pprint
2  print(pprint. doc )
```



That will give you a little bit of documentation attached to the `pprint` function. You can access the same information using the `help` function:

Listing 26.8: `ex26_code.py`

---

```
1  help(pprint)
```



Try these experiments with all of the other dunders you can find. You most likely won’t ever use them directly, but it’s good to know how Python’s internals work.

## Exercise 27. The Five Simple Rules to the Game of Code

---

### Info

This exercise is intended to be studied periodically while you study the next exercises. You're expected to take this very slowly and to mix it with other explanations until you finally get it. If you get lost in this exercise, take a break and do the next ones. Then if you get confused in a later exercise, come back and study the details I describe here. Keep doing this until it "clicks." Remember, you can't fail, so just keep trying until you get it.

---

If you play a game like Go or Chess, you know the rules are fairly simple, yet the games they enable are extremely complex. Really good games have this unique quality of simple rules with complex interactions. Programming is also a game with a few simple rules that create complex interactions, and in this exercise we're going to learn what those rules are.

Before we do that, I need to stress that you most likely won't use these rules directly when you code. There are languages that do utilize these rules directly, and your CPU uses them too, but in daily programming you'll rarely use them. If that's the case, then why learn the rules?

Because these rules are everywhere, and understanding them will help you understand the code you write. It'll help you debug the code when it goes wrong. If you ever want to know how the code works, you'll be able to "disassemble" it down to its basic rules and really see how it works. These rules are a cheat *code*. Pun totally intended.

I'm also going to warn you that **you are not expected to totally understand this right away**. Think of this exercise as setting you up for

the rest of the exercises in this module. You're expected to study this exercise deeply, and when you get stuck, move on to the next exercises as a break. You want to bounce between this one and the next ones until the concepts "click" and they start to make sense. You should also study these rules as deeply as you can, but don't get stuck here. Struggle for a few days, move on, come back, and keep trying. As long as you keep trying, you can't actually "fail."

## Rule 1: Everything Is a Sequence of Instructions

All programs are a sequence of instructions that tell a computer to do something. You've seen Python doing this already when you type code like this:

```
1  x = 10
2  y = 20
3  z = x + y
```

This code starts at line 1, goes to line 2, and so on until the end. That's a sequence of instructions, but inside Python these three lines are converted into another sequence of instructions that look like this:

```
1  LOAD_CONST  0 (10) # load the number 10
2  STORE_NAME  0 (x)  # store that in x
3
4  LOAD_CONST  1 (20) # load the number 20
5  STORE_NAME  1 (y)  # store that in y
6
7  LOAD_NAME    0 (x)  # loads x (which is 10)
8  LOAD_NAME    1 (y)  # loads y (which is 20)
9  BINARY_ADD   # adds those
10 STORE_NAME   2 (z)  # store the result in z
```

That looks totally different from the Python version, but I *bet* you could probably figure out what this sequence of instructions is doing. I've added comments to explain each instruction, and you should be able to connect it back to the previous Python code.

I'm not joking. Take some time right now to connect each line of the Python code to the lines of this "byte code." Using the comments I provided I'm positive you can figure it out, and doing so might turn on a light in your head about the Python code.

It's not necessary to memorize this or even understand each of these instructions. What you should realize is your Python code is being translated into a sequence of simpler instructions that tell the computer to do something. This sequence of instructions is called "byte code" because it's usually stored in a file as a sequence of numbers a computer understands. The output you see above is usually called an "assembly language" because it's a human "readable" (barely) version of those bytes.

These simpler instructions are processed starting at the top, do one small thing at a time, and go to the end when the program exits. That's just like your Python code but with a simpler syntax of INSTRUCTION OPTIONS. Another way to look at this is each part of `x = 10` might become its own instructions in this "byte code."

That's the first rule of The Game of Code: Everything you write eventually becomes a sequence of bytes fed to a computer as instructions for what the computer should do.

## How can I get this output?

To get this output yourself, you use a module called `dis`, which stands for "disassemble." This kind of code is traditionally called "byte code" or "assembly language," so `dis` means to "disassemble." To use `dis` you can import it and use the `dis()` function like this:

```
1  # import the dis function
2  from dis import dis
3
4  # pass code to dis() as a string
5  dis('''
6  x = 10
7  y = 20
```

```
8     z = x + y
9     '''
```

In this Python code I'm doing the following:

1. I import the `dis()` function from the `dis` module
2. I run the `dis()` function, but I give it a multi-line string using `'''`
3. I then write the Python code I want to disassemble into this multi-line string
4. Finally, I end the multi-line string and the `dis()` function with `'''`

When you run this in Jupyter, you'll see it dump the byte code like I have above, but maybe with some extras we'll cover in a minute.

## Where are these bytes stored?

When you run Python (version 3), these bytes are stored in a directory named `__pycache__`. If you put this code into a `ex19.py` file and then run it with `python ex19.py`, you should see this directory.

Looking in this directory you should see a bunch of files ending in `.pyc` with names similar to the code that generated them. These `.pyc` files contain your compiled Python code as bytes.

When you run `dis()`, you're printing a human-readable version of the numbers in the `.pyc` file.

## Rule 2: Jumps Make the Sequence Non-Linear

A sequence of simple instructions like `LOAD_CONST 10` is not very useful. Yay! You can load the number 10! Amazing! Where code starts to become useful is when you add the concept of the "jump" to make this sequence *non-linear*. Let's look at a new piece of Python code:

```
1     while True:
2         x = 10
```



To understand this code we have to foreshadow a later exercise where you learn about the `while`-loop. The code `while True:` simply says “Keep running the code under me `x = 10` while `True` is `True`.” Since `True` will always be `True`, this will loop forever. If you run this in Jupyter, it will never end.

What happens when you `dis()` this code? You see the new instruction `JUMP_ABSOLUTE`:

```
1  dis("while True: x = 10")
2
3      0 LOAD_CONST          1 (10)
4      2 STORE_NAME         0 (x)
5      4 JUMP_ABSOLUTE      0 (to 0)
```

You saw the first two instructions when we covered the `x = 10` code, but now at the end we have `JUMP_ABSOLUTE 0`. Notice there’s numbers `0`, `2`, and `4` to the left of these instructions? In the previous code I cut them out so you wouldn’t be distracted, but here they’re important because they represent locations in the sequence where each instruction lives. All `JUMP_ABSOLUTE 0` does is tell Python to “jump to the instruction at position `0`”, which is `LOAD_CONST 1 (10)`.

With this simple instruction we now have turned boring straight line code into a more complex loop that’s not straight anymore. Later we’ll see how jumps combine with tests to allow even more complex movements through the sequence of bytes.

## Why is this backwards?

You may have noticed that the Python code reads as “while `True` is `True` set `x` equal to `10`” but the `dis()` output reads more like “set `x` equal to `10`, jump to do it again.” That’s because of Rule #1, which says we have to produce a *sequence of bytes only*. There are no nested structures, or any syntax more complex than `INSTRUCTION OPTIONS`, allowed.

To follow this rule Python has to figure out how to translate its code into a sequence of bytes that produces the desired output. That means moving the

actual repetition part to the end of the sequence so it will be in a sequence. You'll find this "backwards" nature comes up often when looking at byte codes and assembly language.

## Can a JUMP go forward?

Yes, technically a JUMP instruction is simply telling the computer to process a different instruction in the sequence. It can be the next one, a previous one, or one in the future. The way this works is the computer keeps track of the "index" of the current instruction, and it simply increments that index.

When you JUMP, you're telling the computer to change this index to a new location in the code. In the code for our while loop (below) the JUMP\_ABSOLUTE is at index 4 (see the 4 to the left). After it runs, the index changes to 0 where the LOAD\_CONST is located, so the computer runs that instruction again. This loops forever.

1	0	LOAD_CONST	1 (10)
2	2	STORE_NAME	0 (x)
3	4	JUMP_ABSOLUTE	0 (to 0)

## Rule 3: Tests Control Jumps

A JUMP is useful for looping, but what about making decisions? A common thing in programming is to ask questions like:

"If x is greater than 0 then set y to 10."

If we write this out in simple Python code, it might look like this:

```
1  if x > 0:
2      y = 10
```

Once again, this is foreshadowing something you'll learn later, but this is simple enough to figure out:

1. Python will *test* if x is greater than > 0

2. If it is, then Python will run the line `y = 10`
3. You see how that line is indented under the `if x > 0:`? That is called a “block” and Python uses indentation to say “this indented code is part of the code above it”
4. If `x` is *NOT* greater than `0`, then Python will *JUMP* over the `y = 10` line to skip it

To do this with our Python byte code we need a new instruction that implements the testing part. We have the JUMP. We have variables. We just need a way to *compare* two things and then a JUMP based on that comparison.

Let’s take that code and `dis()` it to see how Python does this:

```
1  dis('''
2  x = 1
3  if x > 0:
4      y = 10
5  ''')
6
7      0 LOAD_CONST          0 (1)      # load 1
8      2 STORE_NAME         0 (x)      # x = 1
9
10     4 LOAD_NAME          0 (x)      # load x
11     6 LOAD_CONST         1 (0)      # load 0
12     8 COMPARE_OP         4 (>)      # compare x > 0
13    10 POP_JUMP_IF_FALSE  10 (to 20)  # jump if false
14
15    12 LOAD_CONST         2 (10)      # not false, load 10
16    14 STORE_NAME         1 (y)      # y = 10
17    16 LOAD_CONST         3 (None)    # done, load None
18    18 RETURN_VALUE       # exit
19
20    # jump here if false
21    20 LOAD_CONST         3 (None)    # load none
22    22 RETURN_VALUE       # exit
```

The key part of this code is the `COMPARE_OP` and `POP_JUMP_IF_FALSE`:

```
1      4 LOAD_NAME          0 (x)      # load x
2      6 LOAD_CONST        1 (0)      # load 0
3      8 COMPARE_OP        4 (>)      # compare x > 0
4     10 POP_JUMP_IF_FALSE 10 (to 20) # jump if false
```

Here's what this code does:

1. Use `LOAD_NAME` to load the `x` variable
2. Use `LOAD_CONST` to load the `0` constant
3. Use `COMPARE_OP`, which does the `>` comparison and leaves a `True` or `False` result for later
4. Finally, `POP_JUMP_IF_FALSE` makes the `if x > 0` work. It “pops” the `True` or `False` value to get it, and if it reads `False`, it will `JUMP` to instruction 20
5. Doing that will jump over the code that set `y` if the comparison is `False`, *but* if the comparison is `True`, then Python just runs the next instruction, which starts the `y = 10` sequence

Take some time walking through this to try to understand it. If you have a printer, try printing it out and set `x` to different values manually, and then trace through how the code works. What happens when you set `x = -1`?

## What do you mean “pop”?

In the previous code I'm skipping over exactly how Python “pops” the value to read it, but it's storing it in something called a “stack.” For now just think of it as a temporary storage place that you “push” values into and then “pop” them off. You really don't need to go much deeper than that at this stage in your learning. Just understand the effect is to get the result of the last instruction.

## Wait, aren't tests like `COMPARE_OP` used in loops too?

Yes, and you could probably figure out how that works right now based on what you know. Try to write a `while`-loop and see if you can get it to work

with what you know now. Don't worry if you can't though as we'll be covering this in later exercises.

## Rule 4: Storage Controls Tests

You need some way to keep track of changing data while the code operates, and this is done with “storage.” Usually this storage is in the computer's memory and you create names for the data you're storing in memory. You've been doing this when you write code like this:

```
1  x = 10
2  y = 20
3  z = x + y
```

In each of the previous lines we're making a new piece of data and storing it in memory. We're also giving these pieces of memory the names `x`, `y`, and `z`. We can then use these names to “recall” those values from memory, which is what we do in `z = x + y`. We're just recalling the value of `x` and `y` from memory to add them together.

That's the majority of the story, but the important part of this little rule is that you almost always use memory to control tests.

Sure, you can write code like this:

```
1  if 1 < 2:
2      print("but...why?")
```

That's pointless though since it's just running the second line after a pointless test. 1 is always less than 2, so it's useless.

Where tests like `COMPARE_OP` shine is when you use variables to make the tests dynamic based on calculations. That's why I consider this a “rule of The Game of Code” because code without variables isn't really playing the game.

Take the time to go back through the previous examples and identify the places where `LOAD` instructions are used to load values, and `STORE` instructions are used to store values into memory.

## Rule 5: Input/Output Controls Storage

The final rule of The Game of Code is how your code interacts with the outside world. Having variables is great, but a program that has only data you've typed into the source file isn't very useful. What you need is *input* and *output*.

Input is how you get data into your code from things like files, the keyboard, or the network. You've already used `open()` and `input()` to do that in the last module. You accessed input every time you opened a file, read the contents, and did something with them. You also used input when you used ... `input()` to ask the user a question.

Output is how you save or transmit the results of your program. Output can be to the screen with `print()`, to a file with `file.write()`, or even over a network.

Let's run `dis()` on a simple use of `input('Yes? ')` to see what it does:

```
1  from dis import dis
2  dis("input('Yes? ')")
3
4      0 LOAD_NAME          0 (input)
5      2 LOAD_CONST        0 ('Yes? ')
6      4 CALL_FUNCTION      1
7      6 RETURN_VALUE
```

You can see there's now a new instruction `CALL_FUNCTION` that implements the function calls you learned about in [Exercise 18](#). When Python sees `CALL_FUNCTION`, it finds the function that's been loaded with `LOAD_NAME` and then jumps to it to run that function's code. There's a lot more behind how functions work, but you can think of `CALL_FUNCTION` as similar to `JUMP_ABSOLUTE` but to a named place in the instructions.

## Putting It All Together

Taking the five rules, we have the following Game of Code:

1. You read data as input to your program (Rule #5)
2. You store this data in storage (variables) (Rule #4)
3. You use these variables to perform tests... (Rule #3)
4. ... so you can JUMP around... (Rule #2)
5. ... the sequence of instructions... (Rule #1)
6. ... transforming the data into new variables (Rule #4)...
7. ... which you then write to output for storage or display (Rule #5)

While this seems simple, these little rules create very complicated software. Video games are a great example of *very* complicated software that does this. A video game reads your controller or keyboard as input, updates variables that control the models in the scene, and uses advanced instructions that render the scene to your screen as output.

Take the time now to go back through exercises you've completed and see if you understand them better. Does using `dis()` on code you didn't understand help, or is it more confusing? If it helps, then try it on everything to get new insights. If it doesn't help, then just remember it for later. This will be especially interesting when you do it to [Exercise 26](#).

## The List of Byte Codes

As you continue with the exercises I'll have you run `dis()` on some code to analyze what it's doing. You'll need the full list of Python byte codes to study, which can be found [at the end of the `dis\(\)` documentation](#).

### `dis()` Is a Side Quest

Later exercises will have short sections that ask you to run `dis()` on the code to study the byte codes. These sections are “side quests” in your education. That means they are *not* essential for understanding Python, but if you complete them, it may help you later. If they're too hard, then skip them and continue on with the rest of the course.

The most important thing about `dis()` is that it gives you direct access to what *Python* thinks your code does. That can help you if you're confused about how your code works or if you're just curious about what Python is actually doing.



## Exercise 28. Memorizing Logic

Today is the day you start learning about logic. Up to this point you have done everything you possibly can reading and writing files, to the Terminal, and have learned quite a lot of the math capabilities of Python.

From now on, you will be learning *logic*. You won't learn complex theories that academics love to study but just the simple basic logic that makes real programs work and that real programmers need every day.

Learning logic has to come after you do some memorization. I want you to do this exercise for an entire week. Do not falter. Even if you are bored out of your mind, keep doing it. This exercise has a set of logic tables you must memorize to make it easier for you to do the later exercises.

I'm warning you this won't be fun at first. It will be downright boring and tedious, but this teaches you a very important skill you will need as a programmer. You *will* need to be able to memorize important concepts in your life. Most of these concepts will be exciting once you get them. You will struggle with them, like wrestling a squid, and then one day you will understand it. All that work memorizing the basics pays off big later.

Here's a tip on how to memorize something without going insane: Do a tiny bit at a time throughout the day and mark down what you need to work on most. Do not try to sit down for two hours straight and memorize these tables. This won't work. Your brain will retain only whatever you studied in the first 15 or 30 minutes anyway. Instead, create a bunch of index cards with each column on the left (True or False) on the front, and the column on the right on the back. You should then take them out, see the "True or False" and immediately say "True!" Keep practicing until you can do this.

Once you can do that, start writing out your own truth tables each night into a notebook. Do not just copy them. Try to do them from memory. When you get stuck, glance quickly at the ones I have here to refresh your memory. Doing this will train your brain to remember the whole table.

Do not spend more than one week on this, because you will be applying it as you go.

## The Truth Terms

In Python we have the following terms (characters and phrases) for determining if something is “True” or “False.” Logic on a computer is all about seeing if some combination of these characters and some variables is True at that point in the program.

- and
- or
- not
- != (not equal)
- == (equal)
- >= (greater-than-equal)
- <= (less-than-equal)
- True
- False

You actually have run into these characters before but maybe not the terms. The terms (and, or, not) actually work the way you expect them to, just like in English.

## The Truth Tables

We will now use these characters to make the truth tables you need to memorize. First is the table for not x:

NOT	True?
not False	True
not True	False

This is the table for  $x$  or  $y$ :

OR	True?
True or False	True
True or True	True
False or True	True
False or False	False

Now the table for  $x$  and  $y$ :

AND	True?
True and False	False
True and True	True
False and True	False
False and False	False

Then we have the table for not combined with or as not ( $x$  or  $y$ ):

NOT OR	True?
not (True or False)	False
not (True or True)	False
not (False or True)	False
not (False or False)	True

You should compare these tables to the or and and tables to see if you notice a pattern. Here's the table for not ( $x$  and  $y$ ). If you can figure out the pattern, you might not need to memorize them.

NOT AND	True?
not (True and False)	True
not (True and True)	False
not (False and True)	True
not (False and False)	True

Now we get into equalities, which is testing if one thing is equal to another in various ways. First is  $x \neq y$ :

NOT AND	True?
$1 \neq 0$	True
$1 \neq 1$	False
$0 \neq 1$	True
$0 \neq 0$	False

Finally we have  $x == y$ :

NOT AND	True?
$1 == 0$	False
$1 == 1$	True
$0 == 1$	False
$0 == 0$	True

Now use these tables to write up your own cards and spend the week memorizing them. Remember though, there is no failing in this book, just trying as hard as you can each day, and then a *little* bit more.

## Common Student Questions

**Can't I just learn the concepts behind Boolean algebra and not memorize this?** Sure, you can do that, but then you'll have to constantly go through the rules for Boolean algebra while you code. If you memorize these first, not only does it build your memorization skills, but it also makes these operations natural. After that, the concept of Boolean algebra is easy. But do whatever works for you.


## Exercise 29. Boolean Practice

The logic combinations you learned from the previous exercise are called “Boolean” logic expressions. Boolean logic is used *everywhere* in programming. It is a fundamental part of computation, and knowing these logic expressions very well is akin to knowing your scales in music.

In this exercise you will take the logic exercises you memorized and start trying them out in Python. Take each of these logic problems and write what you think the answer will be. In each case it will be either

True or False. Once you have the answers written down, you will start Python in your terminal and type each logic problem in to confirm your answers.

1. `True and True`
2. `False and True`
3. `1 == 1 and 2 == 1`
4. `"test" == "test"`
5. `1 == 1 or 2 != 1`
6. `True and 1 == 1`
7. `False and 0 != 0`
8. `True or 1 == 1`
9. `"test" == "testing"`
10. `1 != 0 and 2 == 1`
11. `"test" != "testing"`
12. `"test" == 1`
13. `not (True and False)`
14. `not (1 == 1 and 0 != 1)`
15. `not (10 == 1 or 1000 == 1000)`
16. `not (1 != 10 or 3 == 4)`
17. `not ("testing" == "testing" and "Zed" == "Cool Guy")`
18. `1 == 1 and (not ("testing" == 1 or 1 == 0))`
19. `"chunky" == "bacon" and (not (3 == 4 or 3 == 3))`
20. `3 != 3 and (not ("testing" == "testing" or "Python" == "Fun`




I will also give you a trick to help you figure out the more complicated ones toward the end.

Whenever you see these Boolean logic statements, you can solve them easily by this simple process:

1. Find an equality test (== or !=) and replace it with its truth
2. Find each and/or inside parentheses and solve those first
3. Find each not and invert it
4. Find any remaining and/or and solve it
5. When you are done, you should have True or False

I will demonstrate with a *variation* on #20:

```
1    3 != 4 and not ("testing" != "test" or "Python" == "Python")
```



Here's me going through each of the steps and showing you the translation until I've boiled it down to a single result:

1. Solve each equality test:
  - `3 != 4` is True, so replace that with True to get True and not (`"testing" != "test" or "Python" == "Python"`)
  - `"testing" != "test"` is True, so replace *that* with True to get True and not (True or `"Python" == "Python"`)
  - `"Python" == "Python"` is True, so replace that with True, and we have True and not (True or True)
2. Find each and/or in parentheses ():
  - (True or True) is True, so replace that to get True and not (True)
3. Find each not and invert it:
  - not (True) is False, so replace that, and we have True and False

4. Find any remaining and/or and solve them:

- True and False is False, and you're done

With that we're done and know the result is False.

---

## Warning!

The more complicated ones may seem very hard at first. You should be able to take a good first stab at solving them, but do not get discouraged. I'm just getting you primed for more of these "logic gymnastics" so that later cool stuff is much easier. Just stick with it, and keep track of what you get wrong, but do not worry that it's not getting in your head quite yet. It'll come.

---

## What You Should See

After you have tried to guess at these, this is what your Jupyter cells might look like:

```
1 >>> True and True
2 True
3 >>> 1 == 1 and 2 == 2
4 True
```

## Study Drills

1. There are a lot of operators in Python similar to != and ==. Try to find as many "equality operators" as you can. They should be like < or <=.
2. Write out the names of each of these equality operators. For example, I call != "not equal."
3. Play with Python by typing out new Boolean operators, and before you press Enter, try to shout out what it is. Do not think about it.

Shout the first thing that comes to mind. Write it down, then press Enter, and keep track of how many you get right and wrong.

4. Throw away the piece of paper from Study Drill 3 so you do not accidentally try to use it later.

## Common Student Questions

**Why does `"test" and "test" return "test" or 1 and 1 return 1` instead of `True`?** Python and many languages like to return one of the operands to their Boolean expressions rather than just `True` or `False`. This means that if you did `False and 1` you get the first operand (`False`), but if you do `True and 1`, you get the second (`1`). Play with this a bit.

**Is there any difference between `!=` and `<>`?** Python has deprecated `<>` in favor of `!=`, so use `!=`. Other than that there should be no difference.

**Isn't there a shortcut?** Yes. Any and expression that has a `False` is immediately `False`, so you can stop there. Any or expression that has a `True` is immediately `True`, so you can stop there. But make sure that you can process the whole expression because later it becomes helpful.



## Exercise 30. What If

Here is the next script of Python you will enter, which introduces you to the `if`-statement. Type this in, make it run exactly right, and then we'll see if your practice has paid off.

Listing 30.1: ex30.py

---

```
1  people = 20
2  cats = 30
3  dogs = 15
4
5
6  if people < cats:
7      print("Too many cats! The world is doomed!")
8
9  if people > cats:
10     print("Not many cats! The world is saved!")
11
12 if people < dogs:
13     print("The world is drooled on!")
14
15 if people > dogs:
16     print("The world is dry!")
17
18
19 dogs += 5
20
21 if people >= dogs:
22     print("People are greater than or equal to dogs.")
23
24 if people <= dogs:
25     print("People are less than or equal to dogs.")
26
27
```

```
28  if people == dogs:
29      print("People are dogs.")
```

## What You Should See

```
1  Too many cats! The world is doomed!
2  The world is dry!
3  People are greater than or equal to dogs.
4  People are less than or equal to dogs.
5  People are dogs.
```

### `dis()` It

For the next few exercises I want you to run `dis()` on some of the code you're studying to get more insight into how it works:

```
1  from dis import dis
2
3  dis('''
4      if people < cats:
5          print("Too many cats! The world is doomed!")
6      ''')
```

This is *not* something you'd do normally when programming. I only want you to do it here to give you one more possible way to understand what's going on. If `dis()` doesn't really help you understand the code more, then feel free to do it and forget it.

To study this, simply put the Python code next to this `dis()` output and try to identify the lines of Python code that match the byte codes.

## Study Drill

In this Study Drill, try to guess what you think the `if`-statement is and what it does. Try to answer these questions in your own words before moving on to the next exercise:

1. What do you think the `if` does to the code under it?
2. Why does the code under the `if` need to be indented four spaces?
3. What happens if it isn't indented?
4. Can you put other Boolean expressions from [Exercise 28](#) in the `if`-statement? Try it.
5. What happens if you change the initial values for `people`, `cats`, and `dogs`?

## Common Student Questions

**What does `+=` mean?** The code `x += 1` is the same as doing `x = x + 1` but involves less typing. You can call this the “increment by” operator. The same goes for `-=` and many other expressions you’ll learn later.

## Exercise 31. Else and If

In the previous exercise you worked out some `if`-statements and then tried to guess what they are and how they work. Before you learn more, I'll explain what everything is by answering the questions you had from Study Drills. You did the Study Drills, right?

1. What do you think the `if` does to the code under it? An `if`-statement creates what is called a “branch” in the code. It's kind of like those choose-your-own-adventure books where you are asked to turn to one page if you make one choice and another if you go a different direction. The `if`-statement tells your script, “If this Boolean expression is `True`, then run the code under it; otherwise skip it.”
2. Why does the code under the `if` need to be indented four spaces? A colon at the end of a line is how you tell Python you are going to create a new “block” of code, and then indenting four spaces tells Python what lines of code are in that block. This is *exactly* the same thing you did when you made functions in the first half of the book.
3. What happens if it isn't indented? If it isn't indented, you will most likely create a Python error. Python expects you to indent *something* after you end a line with a `:` (colon).
4. Can you put other Boolean expressions from [Exercise 28](#) in the `if`-statement? Try it. Yes you can, and they can be as complex as you like, although really complex things generally are bad style.
5. What happens if you change the initial values for people, cats, and dogs? Because you are comparing numbers, if you change the numbers, different `if`-statements will evaluate to `True`, and the blocks of code under them will run. Go back and put different numbers in and see if you can figure out in your head which blocks of code will run.

Compare my answers to your answers, and make sure you *really* understand the concept of a “block” of code. This is important for when you do the next exercise where you write all the parts of if-statements that you can use.

Type this one in and make it work too.

Listing 31.1: ex31.py

```
1  people = 30
2  cars = 40
3  trucks = 15
4
5
6  if cars > people:
7      print("We should take the cars.")
8  elif cars < people:
9      print("We should not take the cars.")
10 else:
11     print("We can't decide.")
12
13 if trucks > cars:
14     print("That's too many trucks.")
15 elif trucks < cars:
16     print("Maybe we could take the trucks.")
17 else:
18     print("We still can't decide.")
19
20 if people > trucks:
21     print("Alright, let's just take the trucks.")
22 else:
23     print("Fine, let's stay home then.")
```

## What You Should See

```
1  We should take the cars.
2  Maybe we could take the trucks.
3  Alright, let's just take the trucks.
```

## `dis()` It

We're now getting to a point where `dis()` is a bit too complicated to study. Let's just pick one of the code blocks to study:

```
1  from dis import dis
2
3  dis('''
4  if cars > people:
5      print("We should take the cars.")
6  elif cars < people:
7      print("We should not take the cars.")
8  else:
9      print("We can't decide.")
10 ''')
```

I think the best way to study this is to put the Python code next to the `dis()` output and try to match the lines of Python to their byte codes. If you can do that, then you're going to be far ahead of many Python programmers who don't even know that Python has `dis()`.

If you can't figure it out, don't worry. It's all about pushing your knowledge as far as possible to find new ways to understand Python.

## Study Drills

1. Try to guess what `elif` and `else` are doing.
2. Change the numbers of `cars`, `people`, and `trucks`, and then trace through each `if`-statement to see what will be printed.
3. Try some more complex Boolean expressions like `cars > people or trucks < cars`.
4. Above each line write an English description of what the line does.

## Common Student Questions

**What happens if multiple `elif` blocks are `True`?** Python starts at the top and runs the first block that is `True`, so it will run only the first one.

## Exercise 32. Making Decisions

In the first half of this book you mostly just printed out things called “functions,” but everything was basically in a straight line. Your scripts ran starting at the top and went to the bottom where they ended. If you made a function, you could run that function later, but it still didn’t have the kind of branching you need to really make decisions. Now that you have `if`, `else`, and `elif` you can start to make scripts that decide things.

In the last script you wrote out a simple set of tests asking some questions. In this script you will ask the user questions and make decisions based on their answers. Write this script, and then play with it quite a lot to figure it out.

Listing 32.1: ex32.py

```
1  print("""You enter a dark room with two doors.
2  Do you go through door #1 or door #2?""")
3
4  door = input("> ")
5
6  if door == "1":
7      print("There's a giant bear here eating a cheese cake.")
8      print("What do you do?")
9      print("1. Take the cake.")
10     print("2. Scream at the bear.")
11
12     bear = input("> ")
13
14     if bear == "1":
15         print("The bear eats your face off. Good job!")
16     elif bear == "2":
17         print("The bear eats your legs off. Good job!")
18     else:
19         print(f"Well, doing {bear} is probably better.")
```



```

19         print("Well, doing {bear} is probably better.")
20     print("Bear runs away.")
21
22     elif door == "2":
23         print("You stare into the endless abyss at Cthulhu's")
24         print("1. Blueberries.")
25         print("2. Yellow jacket clothespins.")
26         print("3. Understanding revolvers yelling melodies.")
27
28     insanity = input("> ")
29
30     if insanity == "1" or insanity == "2":
31         print("Your body survives powered by a mind of")
32         print("Good job!")
33     else:
34         print("The insanity rots your eyes into a pool of")
35         print("Good job!")
36
37 else:
38     print("You stumble around and fall on a knife and d")

```

A key point here is that you are now putting the if-statements *inside* if-statements as code that can run. This is very powerful and can be used to create “nested” decisions, where one branch leads to another and another.

Make sure you understand this concept of if-statements inside if-statements. In fact, do the Study Drills to really nail it.

## What You Should See

Here is me playing this little adventure game. I do not do so well.

```

1  You enter a dark room with two doors.
2  Do you go through door #1 or door #2?
3  > 1
4  There's a giant bear here eating a cheese cake.
5  What do you do?
6  1. Take the cake.

```

```
7  2. Scream at the bear.
8  > 2
9  The bear eats your legs off. Good job!
```

## `dis()` It

There is no `dis()` It section this time because this code is far too complicated to understand, but if you're feeling lucky, then try this:

```
1  from dis import dis
2
3  if door == "1":
4      print("1")
5      bear = input("> ")
6      if bear == "1":
7          print("bear 1")
8      elif bear == "2":
9          print("bear 2")
10     else:
11         print("bear 3")
```

This will produce so much code to analyze, but do the best you can. It does get boring after a while, but it also helps you understand how Python works. Once again, if this is confusing, skip it and try it later.

## Study Drills

1. Make new parts of the game and change what decisions people can make. Expand the game out as much as you can before it gets ridiculous.
2. Write a completely new game. Maybe you don't like this one, so make your own. This is your computer; do what you want.

## Common Student Questions

**Can you replace `elif` with a sequence of `if-else` combinations?** You can in some situations, but it depends on how each `if/else` is written. It also means that Python will check *every* `if-else` combination, rather than just the first false ones like it would with `if-elif-else`. Try to make some of these to figure out the differences.

**How do I tell whether a number is between a range of numbers?**

You have two options: Use  $0 < x < 10$  or  $1 \leq x < 10$ —which is classic notation—or use `x in range(1, 10)`.

**What if I wanted more options in the `if-elif-else` blocks?** Add more `elif` blocks for each possible choice.

## Exercise 33. Loops and Lists

You should now be able to do some programs that are much more interesting. If you have been keeping up, you should realize that now you can combine all the other things you have learned with `if`-statements and Boolean expressions to make your programs do smart things.

However, programs also need to do repetitive things very quickly. We are going to use a `for`-loop in this exercise to build and print various lists. When you do the exercise, you will start to figure out what they are. I won't tell you right now. You have to figure it out.

Before you can use a `for`-loop, you need a way to *store* the results of loops somewhere. The best way to do this is with `lists`. Lists are exactly what their name says: a container of things that are organized in order from first to last. It's not complicated; you just have to learn a new syntax. First, here's how you make lists:

```
1  hairs = ['brown', 'blond', 'red']
2  eyes = ['brown', 'blue', 'green']
3  weights = [1, 2, 3, 4]
```

You start the `list` with the `[` (left bracket), which “opens” the `list`. Then you put each item you want in the list separated by commas, similar to function arguments. Lastly, end the list with a `]` (right bracket) to indicate that it's over. Python then takes this list and all its contents and assigns them to the variable.

---

### Warning!

This is where things get tricky for people who can't code. Your brain has been taught that the world is flat. Remember in the previous exercise where you put `if`-statements inside `if`-statements? That probably made your brain hurt because most

people do not ponder how to “nest” things inside things. In programming nested structures are all over the place. You will find functions that call other functions that have if-statements that have lists with lists inside lists. If you see a structure like this that you can’t figure out, take out a pencil and paper and break it down manually bit by bit until you understand it.

---

We now will build some lists using some for-loops and print them out:

Listing 33.1: ex33.py

```
1  the_count = [1, 2, 3, 4, 5]
2  fruits = ['apples', 'oranges', 'pears', 'apricots']
3  change = [1, 'pennies', 2, 'dimes', 3, 'quarters']
4
5  # this first kind of for-loop goes through a list
6  for number in the_count:
7      print(f"This is count {number}")
8
9  # same as above
10 for fruit in fruits:
11     print(f"A fruit of type: {fruit}")
12
13 # also we can go through mixed lists too
14 for i in change:
15     print(f"I got {i}")
16
17 # we can also build lists, first start with an empty one
18 elements = []
19
20 # then use the range function to do 0 to 5 counts
21 for i in range(0, 6):
22     print(f"Adding {i} to the list.")
23     # append is a function that lists understand
24     elements.append(i)
25
26 # now we can print them out too
27 for i in elements:
```

```
27     for i in elements:
28         print(f"Element was: {i}")
```

## What You Should See

```
1   This is count 1
2   This is count 2
3   This is count 3
4   This is count 4
5   This is count 5
6   A fruit of type: apples
7   A fruit of type: oranges
8   A fruit of type: pears
9   A fruit of type: apricots
10  I got 1
11  I got pennies
12  I got 2
13  I got dimes
14  I got 3
15  I got quarters
16  Adding 0 to the list.
17  Adding 1 to the list.
18  Adding 2 to the list.
19  Adding 3 to the list.
20  Adding 4 to the list.
21  Adding 5 to the list.
22  Element was: 0
23  Element was: 1
24  Element was: 2
25  Element was: 3
26  Element was: 4
27  Element was: 5
```

### **dis()** It

This time let's keep it simple and just see how Python does the for-loop:

```

1  from dis import dis
2
3  dis('''
4  for number in the_count:
5      print(number)
6  ''')
```

This time I'm going to reproduce the output here so we can analyze it:

```

1      0 LOAD_NAME      0 (the_count) # get the count list
2      2 GET_ITER                               # start iteration
3      4 FOR_ITER        6 (to 18)    # for-loop jump to 18
4      6 STORE_NAME      1 (number)    # create number variable
5
6      8 LOAD_NAME      2 (print)      # load print()
7     10 LOAD_NAME      1 (number)     # load number
8     12 CALL_FUNCTION  1              # call print()
9     14 POP_TOP                               # clean stack
10    16 JUMP_ABSOLUTE  2 (to 4)        # jump back to FOR_ITER at 4
11
12    18 LOAD_CONST      0 (None)       # jump here when FOR_ITER done
13    20 RETURN_VALUE
```

Here we see a new thing in the `FOR_ITER` operation. This operation makes the for-loop work by doing these steps:

1. Call `the_count.__next__()`
2. If this says there are no more elements in `the_count`, jump to 18
3. If there are still elements, then continue on
4. The `STORE_NAME` then assigns the result of `the_count.__next__()` to the name `number`

That's all a for-loop actually does. It's mostly a single byte code `FOR_ITER` combined with a few others to iterate through a list.

## Study Drills

1. Take a look at how you used `range`. Look up the `range` function to understand it.
2. Could you have avoided that `for`-loop entirely on line 22 and just assigned `range(0,6)` directly to `elements`?
3. Find the Python documentation on lists and read about them. What other operations can you do to lists besides `append`?

## Common Student Questions

**How do you make a two-dimensional (2D) list?** That's a list in a list like this: `[[1, 2, 3], [4, 5, 6]]`

**Aren't lists and arrays the same thing?** Depends on the language and the implementation. In classic terms a list is very different from an array because of how they're implemented. In Ruby though they call these "arrays." In Python they call them "lists." Just call these "lists" for now since that's what Python calls them.

**Why is a for-loop able to use a variable that isn't defined yet?** The variable is defined by the `for`-loop when it starts, initializing it to the current element of the loop iteration each time through.

**Why does `for i in range(1, 3):` only loop two times instead of three times?** The `range()` function only does numbers from the first to the last, *not including the last*. So it stops at two, not three in the preceding. This turns out to be the most common way to do this kind of loop.

**What does `elements.append()` do?** It simply appends to the end of the list. Open up the Python shell and try a few examples with a list you make. Any time you run into things like this, always try to play with them interactively in the Python shell.



## Exercise 34. While Loops

Now to totally blow your mind with a new loop, the `while`-loop. A `while`-loop will keep executing the code block under it as long as a Boolean expression is `True`.

Wait, you have been keeping up with the terminology, right? That if we write a line and end it with a `:` (colon), then that tells Python to start a new block of code? Then we indent, and that's the new code. This is all about structuring your programs so that Python knows what you mean. If you do not get that idea, then go back and do some more work with `if`-statements, functions, and the `for`-loop until you get it.

Later on we'll have some exercises that will train your brain to read these structures, similar to how we burned Boolean expressions into your brain.

Back to `while`-loops. What they do is simply do a test like an `if`-statement, but instead of running the code block *once*, they jump back to the “top” where the `while` is, and repeat. A `while`-loop runs until the expression is `False`.

Here's the problem with `while`-loops: Sometimes they do not stop. This is great if your intention is to just keep looping until the end of the universe. Otherwise you almost always want your loops to end eventually.

To avoid these problems, there are some rules to follow:

1. Make sure that you use `while`-loops sparingly. Usually a `for`-loop is better.
2. Review your `while`-statements and make sure that the Boolean test will become `False` at some point.
3. When in doubt, print out your test variable at the top and bottom of the `while`-loop to see what it's doing.

In this exercise, you will learn the while-loop while doing these three checks:

Listing 34.1: ex34.py

```
1  i = 0
2  numbers = []
3
4  while i < 6:
5      print(f"At the top i is {i}")
6      numbers.append(i)
7
8      i = i + 1
9      print("Numbers now: ", numbers)
10     print(f"At the bottom i is {i}")
11
12
13     print("The numbers: ")
14
15     for num in numbers:
16         print(num)
```

## What You Should See

```
1  At the top i is 0
2  Numbers now: [0]
3  At the bottom i is 1
4  At the top i is 1
5  Numbers now: [0, 1]
6  At the bottom i is 2
7  At the top i is 2
8  Numbers now: [0, 1, 2]
9  At the bottom i is 3
10 At the top i is 3
11 Numbers now: [0, 1, 2, 3]
12 At the bottom i is 4
13 At the top i is 4
```

```
14  Numbers now: [0, 1, 2, 3, 4]
15  At the bottom i is 5
16  At the top i is 5
17  Numbers now: [0, 1, 2, 3, 4, 5]
18  At the bottom i is 6
19  The numbers:
20  0
21  1
22  2
23  3
24  4
25  5
```

## **dis() It**

For our final “side quest” in The Game of Code you’ll use `dis()` to analyze how a `while`-loop works:

```
1  from dis import dis
2
3  dis('''
4  i = 0
5  while i < 6:
6      i = i + 1
7  ''')
```

You’ve already seen most of these byte codes, so it’s up to you to figure out how this `dis()` output relates to the Python. Remember you can look up all of the byte codes at [the end of the `dis\(\)` documentation](#). Good luck!

## **Study Drills**

1. Convert this `while`-loop to a function that you can call, and replace 6 in the test (`i < 6`) with a variable.
2. Use this function to rewrite the script to try different numbers.

3. Add another variable to the function arguments that you can pass in that lets you change the + 1 on line 8 so you can change how much it increments by.
4. Rewrite the script again to use this function to see what effect that has.
5. Write it to use for-loops and range. Do you need the incrementor in the middle anymore? What happens if you do not get rid of it?

If at any time that you are doing this it goes crazy (it probably will), just hold down CTRL and press c (CTRL-c) and the program will abort.

## Common Student Questions

**What's the difference between a for-loop and a while-loop?** A for-loop can only iterate (loop) “over” collections of things. A while-loop can do any kind of iteration (looping) you want. However, while-loops are harder to get right, and you normally can get many things done with for-loops.

**Loops are hard. How do I figure them out?** The main reason people don't understand loops is because they can't follow the “jumping” that the code does. When a loop runs, it goes through its block of code, and at the end it jumps back to the top. To visualize this, put print statements all over the loop printing out where in the loop Python is running and what the variables are set to at those points. Write print lines before the loop, at the top of the loop, in the middle, and at the bottom. Study the output and try to understand the jumping that's going on.

## Exercise 35. Branches and Functions

You have learned if-statements, functions, and lists. Now it's time to bend your mind. Type this in, and see if you can figure out what it's doing:

Listing 35.1: ex35.py

```
1  from sys import exit
2
3  def gold_room():
4      print("This room is full of gold. How much do you take?")
5
6      choice = input("> ")
7      if "0" in choice or "1" in choice:
8          how_much = int(choice)
9      else:
10         dead("Man, learn to type a number.")
11
12     if how_much < 50:
13         print("Nice, you're not greedy, you win!")
14         exit(0)
15     else:
16         dead("You greedy bastard!")
17
18
19 def bear_room():
20     print("There is a bear here.")
21     print("The bear has a bunch of honey.")
22     print("The fat bear is in front of another door.")
23     print("How are you going to move the bear?")
24     bear_moved = False
25
26     while True:
27         choice = input("> ")
28
29         if choice == "take honey":
```

```

29         if choice == "take money":
30             dead("The bear looks at you then slaps your
31             elif choice == "taunt bear" and not bear_moved:
32                 print("The bear has moved from the door.")
33                 print("You can go through it now.")
34                 bear_moved = True
35             elif choice == "taunt bear" and bear_moved:
36                 dead("The bear gets pissed off and chews your
37             elif choice == "open door" and bear_moved:
38                 gold_room()
39         else:
40             print("I got no idea what that means.")
41
42
43     def cthulhu_room():
44         print("Here you see the great evil Cthulhu.")
45         print("He, it, whatever stares at you and you go insane.")
46         print("Do you flee for your life or eat your head?")
47
48         choice = input("> ")
49
50         if "flee" in choice:
51             start()
52         elif "head" in choice:
53             dead("Well that was tasty!")
54         else:
55             cthulhu_room()
56
57
58     def dead(why):
59         print(why, "Good job!")
60         exit(0)
61
62     def start():
63         print("You are in a dark room.")
64         print("There is a door to your right and left.")
65         print("Which one do you take?")
66
67         choice = input("> ")
68
69

```

```
69     if choice == "left":
70         bear_room()
71     elif choice == "right":
72         cthulhu_room()
73     else:
74         dead("You stumble around the room until you starve.")
75
76
77 start()
```

## What You Should See

Here's me playing the game:

```
1  You are in a dark room.
2  There is a door to your right and left.
3  Which one do you take?
4  > left
5  There is a bear here.
6  The bear has a bunch of honey.
7  The fat bear is in front of another door.
8  How are you going to move the bear?
9  > taunt bear
10 The bear has moved from the door.
11 You can go through it now.
12 > open door
13 This room is full of gold. How much do you take?
14 > 1000
15 You greedy bastard! Good job!
```

## Study Drills

1. Draw a map of the game and how you flow through it.
2. Fix all of your mistakes, including spelling mistakes.

3. Write comments for the functions you do not understand.
4. Add more to the game. What can you do to both simplify and expand it?
5. The `gold_room` has a weird way of getting you to type a number. What are all the bugs in this way of doing it? Can you make it better than what I've written? Look at how `int()` works for clues.

## Common Student Questions

**Help! How does this program work!?** When you get stuck understanding a piece of code, simply write an English comment above *every* line explaining what that line does. Keep your comments short and similar to the code. Then either diagram how the code works or write a paragraph describing it. If you do that, you'll get it.

**Why did you write** `while True`? That makes an infinite loop.

**What does** `exit(0)` **do?** On many operating systems a program can abort with `exit(0)`, and the number passed in will indicate an error or not. If you do `exit(1)`, then it will be an error, but `exit(0)` will be a good exit. The reason it's backward from normal Boolean logic (with `0==False`) is that you can use different numbers to indicate different error results. You can do `exit(100)` for a different error result than `exit(2)` or `exit(1)`.

**Why is** `input()` **sometimes written as** `input('> ')`? The parameter to `input` is a string that it should print as a prompt before getting the user's input.



## Exercise 36. Designing and Debugging

Now that you know `if`-statements, I'm going to give you some rules for `for`-loops and `while`-loops that will keep you out of trouble. I'm also going to give you some tips on debugging so that you can figure out problems with your program. Finally, you will design a little game similar to the previous exercise but with a slight twist.

### From Idea to Working Code

There is a simple process anyone can follow to turn your idea into code. This isn't the *only* process, but it is one that works well for many people. Use this until you develop your own personal process.

1. Get your idea out of your head in any form you understand. Are you a writer? Then write an essay about your idea. Are you an artist or designer? Then draw the user interface. Do you like charts and graphs? Check out the Sequence Diagram, which is one of the most useful diagrams in programming.
2. Create a file for your code. Yes, believe it or not this is an important step that most people stumble over. If you can't come up with a name, just pick a random one for now.
3. Write a description of your idea as comments, in plain English language (or whatever language is easiest for you).
4. Start at the top, and convert the first comment into "pseudo-code," which is kind of Python but you don't care about syntax.
5. Convert that "pseudo-code" into real Python code, and keep running your file until this code does what your comment says.
6. Repeat this until you've converted all of the comments into Python.

7. Step back, review your code, and then *delete it*. You don't have to do this all the time, but if you get in the habit of throwing away your first version, you'll receive two benefits:

- a. Your second version is almost always better than the first.
- b. You confirm to yourself that it wasn't just dumb luck. You actually can code. This helps with impostor syndrome and confidence.

Let's do an example with a simple problem of "create a simple Fahrenheit to Celsius converter." Step 1, I would write out what I know about the conversion:

$C \text{ equals } (F - 32) / 1.8$ . I should ask the user for the F and then print out the C.

A very basic math formula is an easy way to understand the problem. Step 2, I write comments describing what my code should do:

```
1  # ask the user for the F
2  # convert it to a float()
3  # C = (F - 32) / 1.8
4  # print C to the user
```

Once I have that, I "fill in the blanks" with pseudo-code. I'll do just the first line so you can finish this:

```
1  # ask the user for the F
2  F = input(?)
3
4  # convert it to a float()
5  # C = (F - 32) / 1.8
6  # print C to the user
```

Notice I'm being sloppy and not getting the syntax right, which is the point of pseudo-code. Once I have that, convert it to correct Python:

```
1  # ask the user for the F
2  F = input("C? ")
3
4  # convert it to a float()
```

```
5  # C = (F - 32) / 1.8
6  # print C to the user
```

*Run it!* You should be running your code constantly. If you type more than a few lines, just delete them and start over. It's so much easier.

Now that those lines work, I move on to the next comment and repeat the process until I have converted all of the comments into Python. When my script is finally working, I delete it and rewrite it using what I know. Maybe this time I just write the Python directly, or I just repeat the process again. Doing this will confirm to myself that I can actually do it. It was not just dumb luck.

## **Is This a Professional Process?**

You may think that this process is not practical or unprofessional. I think when you're starting out, you need different tools than someone who's been coding for a really long time. I can sit down with an idea and just code, but I've been coding professionally for longer than you may have been alive. Yet, in my head this is essentially the process I follow. I'm just doing it inside my head rapidly, while you have to practice it externally until you internalize it.

I do use this process when I am stuck, or if I'm learning a new language. If I don't know a language but know what I want to do, then I can usually write comments and slowly convert them to code, which also teaches me that language. The only difference between me and you is that I do it faster because of years of training.

## **About the "X/Y" Non-Problem**

Some professionals claim that this process gives students a strange disease called the "X/Y problem." They describe the X/Y problem as "Someone wants to do X, but only knows how to do Y, so they ask for help on how to do Y." The problem with the X/Y problem is it's critical of people who are simply learning how to code and presents no solution. To the "X/Y hater" the solution seems to be "know the answer already," since if they knew how

to do X, they wouldn't bother with Y. The hypocrisy of this belief is that all of the people who hate these kinds of questions also went through a period of doing exactly this and asking these same exact kinds of "X/Y" questions.

The other problem is, they're blaming *you* for their terrible documentation. The classic example is from the original description of the X/Y problem:

```
1  <n00b> How can I echo the last three characters in a filename
2
3  <feline> If they're in a variable: echo ${foo: -3}
4  <feline> Why 3 characters? What do you REALLY want?
5  <feline> Do you want the extension?
6
7  <n00b> Yes.
8
9  <feline> Then ASK FOR WHAT YOU WANT!
10 <feline> There's no guarantee that every filename will
11 have a three-letter extension,
12 <feline> so blindly grabbing three characters does not
13 solve the problem.
14 <feline> echo ${foo##*.}
```

First off, this feline person is literally yelling at someone for asking a question in an IRC channel devoted to answering questions. "ASK FOR WHAT YOU WANT!" The second problem is, their solution is something I—a multi-decade veteran bash and Linux professional—has to look up *every single time*. It is one of the worst documented, least usable features in bash. How is a beginner expected to know ahead of time that they should use some complicated "dollar brace name pound pound asterisk dot brace" operation? This person most likely would not have asked this question had there been simple documentation available online that explained how to do this. Even better would be if bash actually just *had* a basic feature for this incredibly common operation every human needs out of a shell.

When it comes to the "X/Y problem," it is really just an excuse to yell at beginners for being beginners. Every single person who claims to hate this either doesn't actually write code or has *definitely* done exactly this while they were learning to code. That's *how* you learn to code. You come up with

problems and stumble through them learning how to implement solutions. So if you run into someone who acts like <feline>, just ignore them. They're just using you as an excuse to be angry at someone and feel superior.

Additionally, you'll notice that in the previous interaction not a single person *asked to see code*. If <n00b> had just shown their code, then <feline> could have recommended better ways to do that. Problem solved. I mean, assuming <feline> is actually able to code and is not just hanging out in IRC waiting to pounce on unsuspecting beginners asking questions.

## Rules for If-Statements

1. Every `if`-statement must have an `else`.
2. If this `else` should never run because it doesn't make sense, then you must use a `die` function in the `else` that prints out an error message and dies, just like we did in the previous exercise. This will find *many* errors.
3. Never nest `if`-statements more than two deep and always try to do them one deep.
4. Treat `if`-statements like paragraphs, where each `if-elif-else` grouping is like a set of sentences. Put blank lines before and after.
5. Your Boolean tests should be simple. If they are complex, move their calculations to variables earlier in your function and use a good name for the variable.

If you follow these simple rules, you will start writing better code than most programmers. Go back to the previous exercise and see if I followed all of these rules. If not, fix my mistakes.

---

### Warning!

Never be a slave to the rules in real life. For training purposes, you need to follow these rules to make your mind strong, but in real life

sometimes these rules are just stupid. If you think a rule is stupid, try not using it.

---

## Rules for Loops

1. Use a `while`-loop only to loop forever, and that means probably never. This applies only to Python; other languages are different.
2. Use a `for`-loop for all other kinds of looping, especially if there is a fixed or limited number of things to loop over.

## Tips for Debugging

1. Do not use a “debugger.” A debugger is like doing a full-body scan on a sick person. You do not get any specific useful information, and you find a whole lot of information that doesn’t help and is just confusing.
2. The best way to debug a program is to use `print` to print out the values of variables at points in the program to see where they go wrong.
3. Make sure parts of your programs work as you work on them. Do not write massive files of code before you try to run them. Code a little, run a little, fix a little.

## Homework

Now write a game similar to the one that I created in the previous exercise. It can be any kind of game you want in the same flavor. Spend a week on it making it as interesting as possible. For Study Drills, use lists, functions, and modules (remember those from [Exercise 13?](#)) as much as possible, and find as many new pieces of Python as you can to make the game work.

Before you start coding you must draw a map for your game. Create the rooms, monsters, and traps that the player must go through on paper before

you code.

Once you have your map, try to code it up. If you find problems with the map, then adjust it and make the code match.

The best way to work on a piece of software is in small chunks like this:

1. On a sheet of paper or an index card, write a list of tasks you need to complete to finish the software. This is your to-do list.
2. Pick the easiest thing you can do from your list.
3. Write out English comments in your source file as a guide for how you would accomplish this task in your code.
4. Write some of the code under the English comments.
5. Quickly run your script so you can see if that code worked.
6. Keep working in a cycle of writing some code, running it to test it, and fixing it until it works.
7. Cross this task off your list, and then pick your next easiest task and repeat.

This process will help you work on software in a methodical and consistent manner. As you work, update your list by removing tasks you don't really need and adding ones you do.

## **Exercise 37. Symbol Review**

It's time to review the symbols and Python words you know and to try to pick up a few more for the next few lessons. I have written out all the Python symbols and keywords that are important to know.

In this lesson take each keyword and first try to write out what it does from memory. Next, search online for it and see what it really does. This may be difficult because some of these are difficult to search for, but try anyway.

If you get one of these wrong from memory, make an index card with the correct definition and try to “correct” your memory.

Finally, use each of these in a small Python program, or as many as you can get done. The goal is to find out what the symbol does, make sure you got it right, correct it if you did not, and then use it to lock it in.

### **Keywords**



Keyword	Description	Example
and	Logical and	True and False == False
as	Part of the with-as statement	with X as Y: pass
assert	Assert (ensure) that something is true	assert False, "Error!"
break	Stop this loop right now	while True: break
class	Define a class	class Person(object)
continue	Don't process more of the loop; do it again	while True: continue
def	Define a function	def X(): pass
del	Delete from dictionary	del X[Y]
elif	Else if condition	if: X; elif: Y; else: J
else	Else condition	if: X; elif: Y; else: J
except	If an exception happens, do this	except ValueError as e: print(e)
exec	Run a string as Python	exec 'print("hello")'
finally	Exceptions or not, finally do this no matter what	finally: pass
for	Loop over a collection of things	for X in Y: pass
from	Used with import to bring in specific parts of a module	from x import Y

<code>global</code>	Declare that you want a global variable	<code>global X</code>
<code>if</code>	If condition	<code>if: X; elif: Y; else: J</code>
<code>import</code>	Import a module into this one to use	<code>import os</code>
<code>in</code>	Part of for-loops. Also a test of X in Y	<code>for X in Y: pass</code> also <code>1 in [1] == True</code>
<code>is</code>	Like <code>==</code> to test equality	<code>1 is 1 == True</code>
<code>lambda</code>	Create a short anonymous function	<code>s = lambda y: y ** y; s(3)</code>
<code>not</code>	Logical not	<code>not True == False</code>
<code>or</code>	Logical or	<code>True or False == True</code>
<code>pass</code>	This block is empty	<code>def empty(): pass</code>
<code>print</code>	Print this string	<code>print('this string')</code>
<code>raise</code>	Raise an exception when things go wrong	<code>raise ValueError("No")</code>
<code>return</code>	Exit the function with a return value	<code>def X(): return Y</code>
<code>try</code>	Try this block, and if exception, go to except	<code>try: pass</code>
<code>while</code>	While loop	<code>while X: pass</code>
<code>with</code>	With an expression as a variable do	<code>with X as Y: pass</code>
<code>yield</code>	Pause here and return to caller	<code>def X(): yield Y; X().next()</code>

## Data Types

For data types, write out what makes up each one. For example, with strings, write out how you create a string. For numbers, write out a few numbers.

Type	Description	Example
True	True boolean value	True or False == True
False	False boolean value	False and True == False
None	Represents "nothing" or "no value"	x = None
bytes	Stores bytes, maybe of text, PNG, file, etc.	x = b"hello"
strings	Stores textual information	x = "hello"
numbers	Stores integers	i = 100
floats	Stores decimals	i = 10.389
lists	Stores a list of things	j = [1,2,3,4]
dicts	Stores a key=value mapping of things	e = {'x': 1, 'y': 2}

## String Escape Sequences

For string escape sequences, use them in strings to make sure they do what you think they do.

Escape	Description
\\	Backslash
\'	Single-quote
\"	Double-quote
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline
\r	Carriage
\t	Tab
\v	Vertical tab

## Old-Style String Formats

It's the same thing for string formats: use them in some strings to know what they do.

Escape	Description	Example
%d	Decimal integers (not floating point)	"%d" % 45 == '45'
%i	Same as %d	"%i" % 45 == '45'
%o	Octal number	"%o" % 1000 == '1750'
%u	Unsigned decimal	"%u" % -1000 == '-1000'
%x	Hexadecimal lowercase	"%x" % 1000 == '3e8'
%X	Hexadecimal uppercase	"%X" % 1000 == '3E8'
%e	Exponential notation, lowercase 'e'	"%e" % 1000 == '1.000000e+03'
%E	Exponential notation, uppercase 'E'	"%E" % 1000 == '1.000000E+03'
%f	Floating point real number	"%f" % 10.34 == '10.340000'
%F	Same as %f	"%F" % 10.34 == '10.340000'
%g	Either %f or %e, whichever is shorter	"%g" % 10.34 == '10.34'
%G	Same as %g but uppercase	"%G" % 10.34 == '10.34'
%c	Character format	"%c" % 34 == ''
%r	Repr format (debugging format)	"%r" % int == "<type 'int'>"
%s	String format	"%s there" % 'hi' == 'hi there'
%%	A percent sign	"%g%" % 10.34 == '10.34%'

Older Python 2 code uses these formatting characters to do what f-strings do. Try them out as an alternative.

## Operators

Some of these may be unfamiliar to you, but look them up anyway. Find out what they do, and if you still can't figure it out, save it for later.

Escape	Description	Example
%d	Decimal integers (not floating point)	"%d" % 45 == '45'
%i	Same as %d	"%i" % 45 == '45'
%o	Octal number	"%o" % 1000 == '1750'
%u	Unsigned decimal	"%u" % -1000 == '-1000'
%x	Hexadecimal lowercase	"%x" % 1000 == '3e8'
%X	Hexadecimal uppercase	"%X" % 1000 == '3E8'
%e	Exponential notation, lowercase 'e'	"%e" % 1000 == '1.000000e+03'
%E	Exponential notation, uppercase 'E'	"%E" % 1000 == '1.000000E+03'
%f	Floating point real number	"%f" % 10.34 == '10.340000'
%F	Same as %f	"%F" % 10.34 == '10.340000'
%g	Either %f or %e, whichever is shorter	"%g" % 10.34 == '10.34'
%G	Same as %g but uppercase	"%G" % 10.34 == '10.34'
%c	Character format	"%c" % 34 == ''
%r	Repr format (debugging format)	"%r" % int == "<type 'int'>"
%s	String format	"%s there" % 'hi' == 'hi there'
%%	A percent sign	"%g%" % 10.34 == '10.34%'

Spend about a week on this, but if you finish faster, that's great. The point is to try to get coverage on all these symbols and make sure they are locked in your head. What's also important is to find out what you *do not* know so you can fix it later.

## Reading Code

Now find some Python code to read. You should be reading any Python code you can and trying to steal ideas that you find. You actually should have enough knowledge to be able to read but maybe not understand what the code does. What this lesson teaches is how to apply things you have learned to understand other people's code.

First, print out the code you want to understand. Yes, print it out, because your eyes and brain are more used to reading paper than computer screens. Make sure you print a few pages at a time.

Second, go through your printout and take notes on the following:

1. Functions and what they do.
2. Where each variable is first given a value.
3. Any variables with the same names in different parts of the program. These may be trouble later.
4. Any `if`-statements without `else` clauses. Are they right?
5. Any `while`-loops that might not end.
6. Any parts of code that you can't understand for whatever reason.

Third, once you have all of this marked up, try to explain it to yourself by writing comments as you go. Explain the functions, how they are used, what variables are involved and anything you can to figure this code out.

Lastly, on all of the difficult parts, trace the values of each variable line by line, function by function. In fact, do another printout, and write in the margin the value of each variable that you need to “trace.”

Once you have a good idea of what the code does, go back to the computer and read it again to see if you find new things. Keep finding more code and doing this until you do not need the printouts anymore.

## Study Drills

1. Find out what a “flow chart” is and draw a few.
2. If you find errors in code you are reading, try to fix them, and send the author your changes.
3. Another technique for when you are not using paper is to put `#` comments with your notes in the code. Sometimes, these could become the actual comments to help the next person.

## Common Student Questions

**How would I search for these things online?** Simply put “python3” before anything you want to find. For example, to find `yield` search for `python3 yield`.

## **Module 3: Applying What You Know**



## **Exercise 38. Beyond Jupyter for Windows**

**[This content is currently in development.]**

**This content is currently in development.**

## **Exercise 39. Beyond Jupyter for macOS/Linux**

**[This content is currently in development.]**

**This content is currently in development.**

## **Exercise 40. Advanced Developer Tools [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 41. A Project Skeleton [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 42. Doing Things to Lists [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 43. Doing Things to Dictionaries [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 44. From Dictionaries to Objects [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 45. Basic Object-Oriented Programming [This content is currently in development.]**

**This content is currently in development.**



## **Exercise 46. Inheritance and Advanced OOP**

**[This content is currently in development.]**

**This content is currently in development.**

## **Exercise 47. Basic Object-Oriented Analysis and Design [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 48. Inheritance versus Composition**

**[This content is currently in development.]**

**This content is currently in development.**

## **Exercise 49. You Make a Game [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 50. Automated Testing [This content is currently in development.]**

**This content is currently in development.**

## **Module 4: Python and Data Science**

## **Exercise 51. What Is Data Munging? [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 52. Scraping Data from the Web [This content is currently in development.]**

**This content is currently in development.**



## **Exercise 53. Getting Data from APIs [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 54. Data Conversion with Pandas**

**[This content is currently in development.]**

**This content is currently in development.**

## **Exercise 55. How to Read Documentation (Featuring Pandas) [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 56. Using Only Pandas [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 57. The SQL Crash Course [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 58. SQL Normalization [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 59. SQL Relationships [This content is currently in development.]**

**This content is currently in development.**

## **Exercise 60. Advice from an Even Older Programmer [This content is currently in development.]**

**This content is currently in development.**