
Arcade Documentation

Release Spring 2019

Paul Vincent Craven

Apr 24, 2019

Contents

1	Chapters	1
1.1	Foreword - Why Learn Programming?	1
1.2	What is a Programming Language?	2
1.3	Setting Up Your System	7
1.4	Version Control Systems	8
1.5	The <code>print</code> Function	24
1.6	How to Draw with Your Computer	26
1.7	Variables and Expressions	36
1.8	Creating Functions	42
1.9	Drawing With Functions	54
1.10	If Statements	62
1.11	Guessing Games with Random Numbers and Loops	72
1.12	Advanced Looping	89
1.13	Introduction to Lists	96
1.14	Introduction to Classes	108
1.15	Using the Window Class	130
1.16	User Control	137
1.17	Sound Effects	150
1.18	Sprites And Collisions	152
1.19	Moving Sprites	165
1.20	Debugging Programs	178
1.21	Using Sprites to Shoot	178
1.22	Sprites and Walls	189
1.23	Libraries and Modules	205
1.24	Searching	221
1.25	Array-Backed Grids	237
1.26	Platformers	247
1.27	Sorting	257
1.28	Exceptions	262
1.29	Recursion	267
1.30	String Formatting	284
1.31	Setting Up a Virtual Environment In PyCharm	295
2	Labs	297
2.1	Lab 1: First Program	297
2.2	Lab 2: Draw a Picture	298

2.3	Lab 3: Drawing with Functions	301
2.4	Lab 4: Camel	303
2.5	Lab 5: Loopy Lab	307
2.6	Lab 6: Text Adventure	312
2.7	Lab 7: User Control	317
2.8	Lab 8: Sprites	318
2.9	Lab 9: Sprites and Walls	319
2.10	Lab 10: Spell Check	320
2.11	Lab 11: Array-Backed Grids	322
2.12	Lab 12: Final Lab	324
2.13	Quantitative Reasoning Reflection	328

CHAPTER 1

Chapters

1.1 Foreword - Why Learn Programming?

Why program? Programming is:

- Fun
- Pays well

1.1.1 Programming Is Fun

With programming you get to *create*. People paint and perform music to create and express themselves.

Picasso had a saying. He said, “Good artists copy, great artists steal.” And we have always been shameless about stealing great ideas and I think part of what made the Macintosh great was that the people working on it were musicians and poets and artists and zoologists and historians who also happened to be the best computer scientists in the world.

Steve Jobs in PBS's “Triumph of the Nerds: The Rise of Accidental Empires” (1996)

Not only do you get to be an artist when you program, but you get to design.

Design is a funny word. Some people think design means how it looks. But of course, if you dig deeper, it's really how it works. The design of the Mac wasn't what it looked like, although that was part of it. Primarily, it was how it worked.

Steve Jobs in Wired's “The Next Insanely Great Thing” (1996)

1.1.2 Programming Pays Well

Search up “top paid majors” and you'll see computer science is usually one of the top paid degrees you can get.

The National Association of Colleges and Employers (NACE) says that Computer Science is the second best paid degree (as of Jan 2018) that you can get. Engineering is number one.

Payscale breaks the engineering degrees out, and puts computer science at #14. The other top-paid majors, such as engineering and mathematics, all use programming extensively.

In addition we'll teach the topics of a typical first-semester programming class. But the most important thing is for you to find out if programming is worth your time.

Note: In *How to Draw with Your Computer* we show how to create your first graphics. These first chapters we show you how to get your computer set up.

1.1.3 What Other Students Have Made

Here are some videos of games students have created in prior semesters. Students who didn't know any programming, were able to create these games by the time they finished the semester.

1.2 What is a Programming Language?

What is a programming language? This wouldn't be much of a programming course if you left without even knowing what a programming language was! So let's get that out of the way.

Note: The CPU is the “brain” of the computer.

Computers have a chip called the Central Processing Unit ([CPU](#)) that functions as the main “brain” of the computer. For example, right now you might have an Intel i7 or an AMD-FX CPU in your computer.



Fig. 1: Intel i7 CPU ([Wikipedia Commons: CPU](#))

The CPU gets its instructions by reading a set of numbers. For example the number “04” might be an instruction to add two other numbers together.

Everything stored on the computer is in the form of numbers. Some numbers computers store are for data (text, photos, movies), and some are computer instructions.

1.2.1 Machine Code

In the early days of computing, programmers punched in numbers that represented commands for the CPU. Then the programmers punched in the data.

Note: Machine code is the native language of any computer.

We call these numbers that are instructions [machine code](#). All machine code is made of numbers, but not all numbers are machine code. Some of the numbers might be data to hold text or images. Machine code is also called a [First Generation Language](#) (1GL).

Below is an image of the [Altair 8800](#), the first personal computer that regular people could buy. Notice that it is missing a monitor and a keyboard! The first computers loaded the instructions by flipping switches. A pattern of switches represented a machine instruction. So you'd flip lots of switches, then hit the "Run" switch. And the lights would blink.



Fig. 2: Source: [Wikipedia: Altair 8800](#)

While this may not seem very useful (and quite frankly, it wasn't) it was very popular in the hobbyist community. Those people saw the potential.

Computers *still* run on machine code. You can still code by punching in numbers if you want. But you'd be crazy because hand-coding these numbers is *so* tedious. There's something better. Assembly Language.

1.2.2 Assembly Language

In order to make things easier, computer scientists came up with something called [assembly language](#). Assembly language is a [Second Generation Language](#) (2GL). Assembly language looks like this:

Don't worry! We aren't coding in assembly language for this class.

Assembly language allows a programmer to edit a file and type in codes like `LDA` which stands for "Load Accumulator Immediate." The programmer types these commands into a [source file](#). We call the commands [source code](#). The computer can't run the source code as-is. The programmer runs a [compiler](#) that simply translates the computer commands like `LDA` into the corresponding number of the machine language instruction.

```

MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER PAGE    2

C000          ORG      ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START    LDS      #STACK

*****
* FUNCTION: INITA - Initialize ACIA
* INPUT: none
* OUTPUT: none
* CALLS: none
* DESTROYS: acc A

0013        RESETA   EQU      %00010011
0011        CTLREG   EQU      %00010001

C003 86 13  INITA    LDA A  #RESETA  RESET ACIA
C005 B7 80 04          STA A  ACIA
C008 86 11          LDA A  #CTLREG  SET 8 BITS AND 2 STOP
C00A B7 80 04          STA A  ACIA

C00D 7E C0 F1        JMP     SIGNON   GO TO START OF MONITOR

*****
* FUNCTION: INCH - Input character
* INPUT: none
* OUTPUT: char in acc A
* DESTROYS: acc A
* CALLS: none
* DESCRIPTION: Gets 1 character from terminal

C010 B6 80 04  INCH    LDA A  ACIA      GET STATUS
C013 47          ASR A           SHIFT RDRE FLAG INTO CARRY
C014 24 FA          BCC   INCH      RECEIVE NOT READY
C016 B6 80 05          LDA A  ACIA+1  GET CHAR
C019 84 7F          AND A  #$7F    MASK PARITY
C01B 7E C0 79          JMP   OUTCH   ECHO & RTS

*****
* FUNCTION: INHEX - INPUT HEX DIGIT
* INPUT: none
* OUTPUT: Digit in acc A
* CALLS: INCH
* DESTROYS: acc A
* Returns to monitor if not HEX input

C01E 8D F0  INHEX    BSR     INCH      GET A CHAR
C020 81 30          CMP A  #'0    ZERO
C022 2B 11          BMI    HEXERR   NOT HEX
C024 81 39          CMP A  #'9    NINE
C026 2F 0A          BLE    HEXRTS   GOOD HEX
C028 81 41          CMP A  #'A    NOT HEX
C02A 2B 09          BMI    HEXERR   NOT HEX
C02C 81 46          CMP A  #'F    NOT HEX
C02E 2E 05          BGT    HEXERR   NOT HEX
C030 80 07          SUB A  #7    FIX A-F
C032 84 0F  HEXRTS  AND A  #$0F    CONVERT ASCII TO DIGIT
C034 39          RTS           RTS

C035 7E C0 AF  HEXERR  JMP     CTRL     RETURN TO CONTROL LOOP

```

Fig. 3: Source: Wikipedia: Motorola 6800 Assembly Language

Note: A compiler turns human-readable code into machine code.

After I compile the source code into compiled code, I can run the compiled code. I can give the compiled code to someone else and they can run it. They do not need the source code or the compiler.

Assembly language is an improvement over machine language. But it isn't *that* much of an improvement. Why? Assembly language instructions are very low-level. There are no commands like "draw a building here." Or even "print hi." There are only mind-numbingly simple commands that move bits from one spot to another, add them, and shift them.

1.2.3 Third Generation Languages



Fig. 4: Source: Wikipedia Grace Hopper and UNIVAC

Third Generation Languages (3GL) started with Grace Hopper creating the language **COBOL**. There are many, many different third generation languages now. These languages often specialize at certain tasks. For example, the language **C** is great at creating small, fast programs that can run on minimal hardware. **PHP** is an easy-to-use language that can build websites.

Note: Most of the original computer scientists were female. See [Grace Hopper](#), [Hedy Lamar](#), and [Ada Lovelace](#) for examples. If you want to find other female programmers who code in Python, check out [@PyLadies](#), [@DjangoGirls](#), and [@WomenWhoCode](#).

Third generation languages usually fall into one of three categories.

- **Compiled:** The computer takes the original source code, and uses a *compiler* to translate it to machine code. The user then runs the machine code. The original source code is not needed to run the program. "C" is an example of a language that works this way. So is the 2GL assembly language we just talked about.

- **Interpreted:** The computer looks at the source code and translates/runs it line-by-line. The compile step is not needed, but the user needs both the source code and an interpreter to run the program. Python is an example of an interpreted language.
- **Runtime Environment:** Languages such as Java and C# take source code, and compile the source code to a machine language. But not the language of your actual machine, they compile to a *virtual* machine. This is a separate program that acts as a layer between the real machine and the compiled code. This allows for better security, portability, and memory management.

Working with a compiled language is like taking a book in Spanish and translating it to English. You no longer need the Spanish book, and you don't need the translator. However, if you want to edit or change the book you have to re-translate everything.

Working with an interpreted language is like working with a interpreter. You can communicate back and forth with a person that knows both English and Spanish. You need the original Spanish, the English, and the interpreter. It is easier to make ad-hoc changes and carry out a dialog. Interpreters often help prevent computers from running commands that will cause major crashes or common security issues. Kind of like having a human interpreter that says, "You don't *really* want to say that."

Using a runtime environment is hard to explain in human terms. It is a hybrid of the two system. You need source code. You need a compiler. Instead of the compiler making machine code, it makes for for a **virtual machine**.

1.2.4 What is so great about Python?

Python is a great language to start programming in. Python is a Top-5 language in popularity according to the [TIOBE Index](#). While may be less popular than Java, it is easier to read and learn. Less work is required to do graphics. And everything you learn in Python you can also apply when you learn [C#](#) or [Java](#).

Python a great language for people interested in [automating boring things](#). Python is also extremely popular in data analytics. Typically researchers will use the add-ons [Pandas](#) and [Jupyter Notebooks](#).

1.2.5 Python 2.7 vs. Python 3.7

There are two main versions of Python. When Python moved to version 3, there were changes that didn't work with all the currently written Python 2 programs. So both Python 2 and Python 3 were being developed simultaneously. Some people don't want to move to Python 3 at all.

We use Python 3. Why are you going to care?

- If you search up examples you will find both Python 2 and Python 3 examples.
- Systems such as the Mac and Linux have Python 2 installed by default.

If you see a Python example on the web that has a print statement that looks like:

```
# A "print" statement with Python Version 2.x
print "Hi"
```

Instead of:

```
# A "print" statement with Python Version 3.x
print("Hi")
```

Then you have a Python 2 example and it won't run with what we install and use in this class.

In the case of the Mac and Linux, it will be important to use Python 3 and not Python 2. Since Python 2 is installed by default, it can be a bit of a hassle to make sure they use Python 3.

1.3 Setting Up Your System

Before you begin, you need to install a few things on the computer. Using a school computer? You may have these already installed. Yay! Still, read through this part. Make sure you know how the computer is setup up.

Getting a system set up and ready to program can be a bit frustrating. **Don't give up.** Once you've got a system set up for programming you don't need to think about it again. You can concentrate on the programming part. You may get stuck during the setup, so don't hesitate to reach out and get help.

Your development computer will need:

- The [Python](#) programming language and the [Arcade](#) code library.
- An editor to type in your programs. (We'll use a program called [PyCharm](#). You can use the community edition for free, or if you have an email address that ends in `.edu` you can get a free [student license](#) for the professional version.)
- A version control system to track and turn in your work. ([Git/SourceTree/BitBucket](#))
- A minor configuration tweak so we can see file extensions.

Let's go through these four items in detail.

1.3.1 Setup the Programming Environment

To get your computer ready for programming, we need to install Python, some Python libraries, and an editor.

Installing Python and Arcade

We will be using the “[Python](#)” computer programming language. The creator of Python was a fan of [Monty Python](#), hence the name.

In addition to the Python language, we are going to use a library of commands for drawing on the screen. This is called the “[Arcade](#)” library.

Installation for installing Python and the Arcade library are available below:

- [Windows Installation](#) (Make sure to read the instructions carefully. Do not skip the “Add Python to Path” step. This seems to be the most frequent issue.)
- [Mac Installation](#)
- [Linux Installation](#)

Installing an IDE

We also need an editor. Python comes with an editor called IDLE, but it is awful and not worth using. We'll use an editor called PyCharm.

PyCharm is a powerful program that lets you do more than just edit the program, it also includes a large set of tools that programmers need. This type of environment is called an **Integrated Development Environment**, or **IDE** for short.

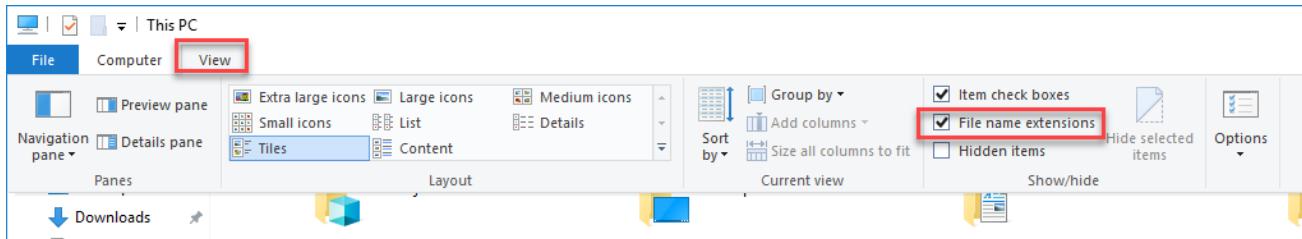
Download and install [PyCharm](#). You can use their community edition for free. We won't use the features in the professional edition. If you decide to pick the professional edition anyway, you'll need a license to use it. But licenses are free for educational use! If you have an e-mail that ends in `.edu` you can ask for a [student license](#). It can be used on a school computer, or on your own computer.

What is a text editor? What is an IDE? Read more at [Understanding and Choosing Text Editors](#).

1.3.2 Viewing File Extensions

It is a great idea to change your windows configuration to show file extensions. A file usually has a name like `Book report.docx` where the `.docx` tells the computer it is a Microsoft Word compatible document. By default Windows hides the `.docx` extension if there is a program installed to handle it. If you are programming, this hiding part of the file name can be annoying.

For Windows 8 and 10, bring up a file explorer by hitting the Windows-E key. Then click the “view” tab and make sure “File name extensions” has been checked.



1.4 Version Control Systems

We will use a *version control system* to track your work, and allow the instructor to give feedback. Odds are, this will be the same exact system you'd use in a large company.

1.4.1 Quick Reference

Commit and Upload

Do this whenever you are done with a session of programming:

1. Open “git bash” on Windows, or “terminal” on MacOS.
2. Type `cd mydirectoryname` where the name of the directory will be the same as the name of your repository on BitBucket. You can usually type in the first few letters and hit `<tab>` to fill in the rest of the directory name.
3. Type `git add *`
4. Type `git commit -m "Work on lab 1"` Update the comment between the quotes to whatever you did.
5. Type `git push`

If you get an error while pushing, see [What If You Can't Push?](#)

Turn In Your Work

1. Go to BitBucket
2. Click on “Source”
3. Find the folder with your lab
4. Copy link
5. Go to Scholar for the lab

6. Paste link, and turn in.

For more commands, see the [Longer Git Command Reference](#).

1.4.2 What is a Distributed Version Control System

No serious development should be done without version control. In fact, version control is so important, many developers would argue that almost no development should be done without version control. Even all my notes for class I keep in version control.

Version control allows developers to:

- Get any prior version of a project.
 - Released version 1.5 of your program, and now it is crashing? Quick! Go back to version 1.4.
 - Did the ‘new guy’ mess up the project? Revert back!
- Know exactly what changed in the code, when, and by who. See who is actually doing the work. If a mistake gets added in, see when it was added and by whom.
- Easily share code between developers.
- Easily work independently of other developers.
- Recover an accidentally deleted or overwritten file.
- Go back and create a bug-fix release on prior versions of a program.
- Work on multiple computers and keep files in sync.

Version control saves untold time and headaches. It used to be that version control had enough of a learning curve that some developers refused to use it. Thankfully today’s version control tools are so easy to use there’s no excuse not to.

There are two main types of version control. The original version control systems were “centralized.” Subversion (SVN) is a very popular piece of software that supports this type of version control. The other type is a “Distributed Version Control Systems” (DVCS). There are two popular versions of DVCS in use today, [Git](#) and [Mercurial](#). Mercurial is sometimes also known as Hg. Get it? Hg is the symbol for Mercury. Either Git or Hg works fine, but for this tutorial we will standardize on Git.

1.4.3 Installing Git

Let’s install `git` on your computer. If you are using a school computer with `git` pre-installed, you can skip this step.

Click the link below and download and install the 64-bit version of the `git`.

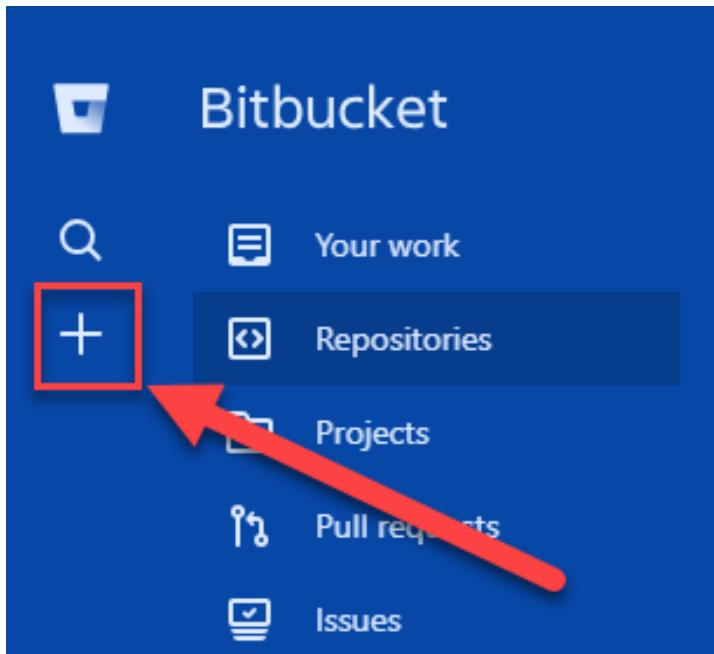
- [Windows Git DVCS](#)
- [MacOS Git DVCS](#)

1.4.4 Forking the Repository

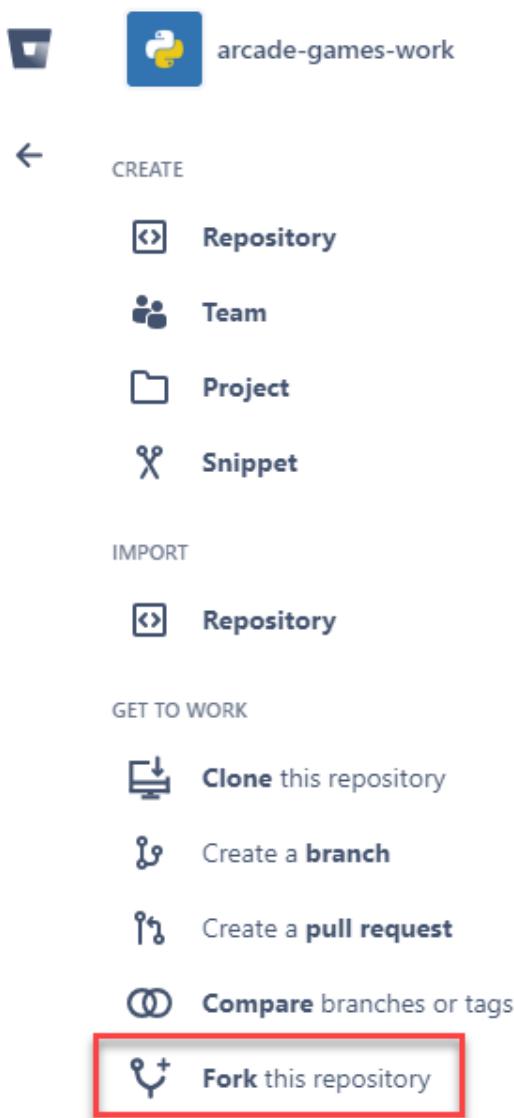
Attention: You should only have to fork the code **once** during class. If you do it more than once, something is wrong. Stop before you do this and see the instructor. It is a big headache for everyone if you fork more than once.

1. We are going to store our programs on-line with a website called BitBucket. BitBucket and a program called SourceTree are owned by a company called Atlassian. They offer enhanced accounts for e-mail addresses ending in `.edu`. To use BitBucket, create an account <https://bitbucket.org/account/signup/>

2. Go to this web address which has a template for the labs we'll create in class: <https://bitbucket.org/pcraven/arcade-games-work>
3. We need to “fork” the repository. This will create your own copy of the repository that will be independent of mine. Changes you make to a “fork” aren’t automatically sent to the original. Fork the repository by clicking on the plus button:



4. Then select “Fork”:



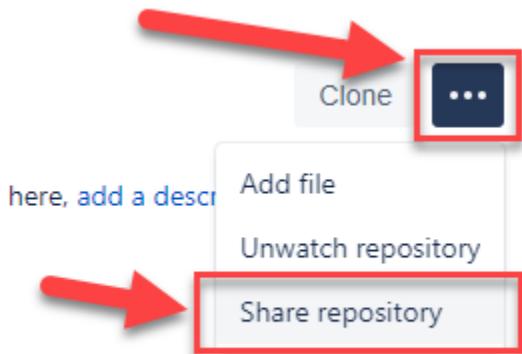
5. Next, select a name for your fork. Use your last name and first name. Also, select that your repository is private, so that you don't share your homework answers with the world.



6. Now you have your own fork. It exists on the BitBucket server only.

1.4.5 Share the Repository

1. Give read permission to the instructor pcraven for your fork so he can grade your assignments.



Then

User and group access

Here's where you grant users and **groups** access to this repository. For a [list of all users](#) repositories, see which users count towards your bill on the [Users on plan page](#). [Learn more](#)

Users

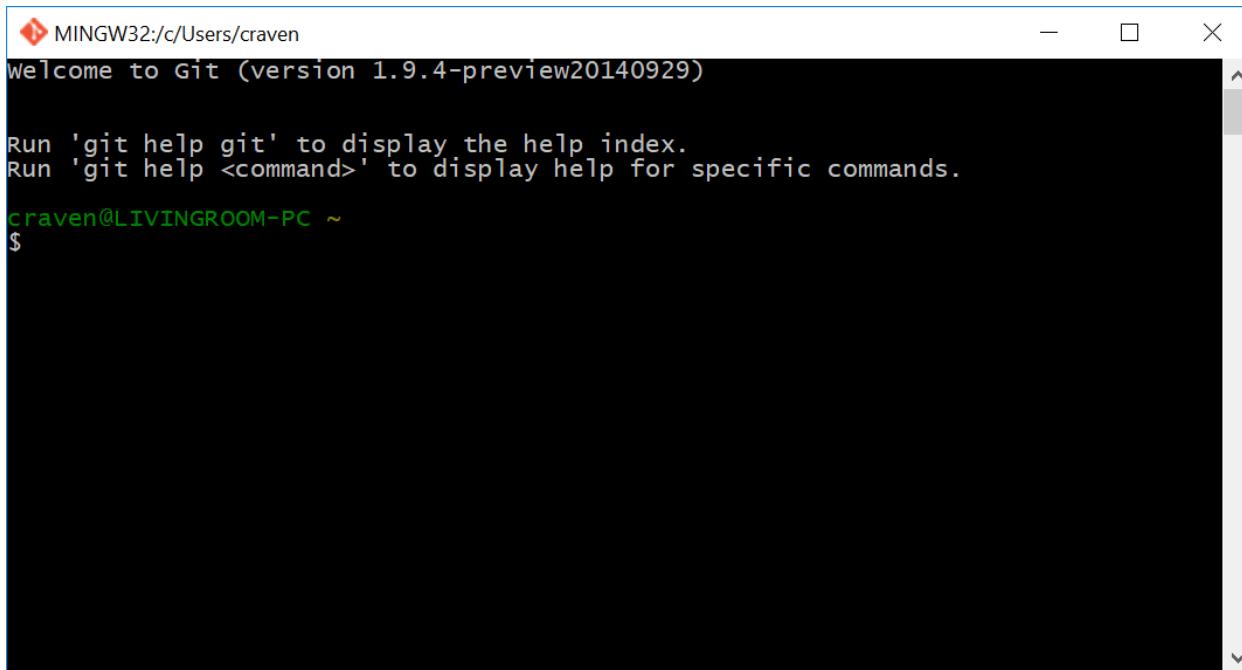
The screenshot shows a search bar with 'pcraven' typed in. Below it, a list shows a single result: 'Paul Craven pcraven' with a profile icon.

Attention: Check the feedback on your homework. If the instructor can't get to your homework, you'll get a zero. You need to correct this and resubmit ASAP.

1.4.6 Cloning the Repository

Note: Every time you start working on a new computer, you'll need to create a clone. (Unless you use a flash drive.)

1. Run the program “Git Bash” on Windows. Or, if you are on the mac, go under “Applications”, find “Utilities” and in that run “Terminal”.



A screenshot of a Git Bash window. The title bar says "MINGW32:c/Users/craven". The window content shows the following text:
Welcome to Git (version 1.9.4-preview20140929)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

craven@LIVINGROOM-PC ~
\$

Fig. 5: Git Bash Window

2. Figure out where you want to store your files. You might want to store the files on your laptop, a flash drive, or a networked drive.
3. Figure out what directory your “Bash” window is in. Do this by typing `pwd`, which is short for “print working directory”.

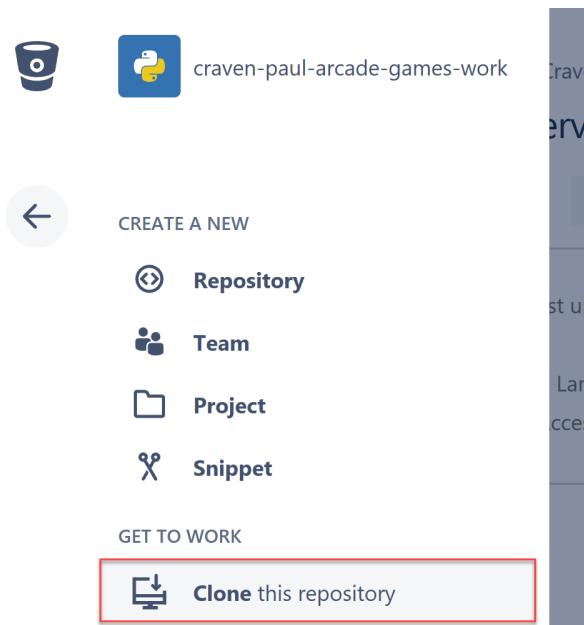
```
MINGW32:c/Users/craven
Welcome to Git (version 1.9.4-preview20140929)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

craven@LIVINGROOM-PC ~
$ pwd
/c/users/craven

craven@LIVINGROOM-PC ~
$
```

4. You can see what files are in the directory by typing `ls`, short for “list files”.
5. You can change directories using the `cd` command. You should default to your “home” directory, which is a great place to put your files. But if you want them in a different location, change to that location now. There’s a lot to the `cd` command, but there are a few variations you need to know:
 - `cd` Change to your “home” directory.
 - `cd mydir` Change to `mydir` directory. That directory must be in the same directory you are in now. Also, if you don’t want to type the full directory name, you can type the first few letters and hit `<tab>`.
 - `cd ..` Go up one directory.
6. We want to copy the repository you created to your computer. We’ll call this a “clone.” A “clone” is a copy we normally try to keep synced up, which is different than a “fork.” To clone the repository, hit the “plus” and then select “Clone Repository”



7. Copy the address that it gives you. It should have **your** name, and **not** my name. If you get this wrong, you'll have to restart everything back at the clone section. (Not the fork section.)



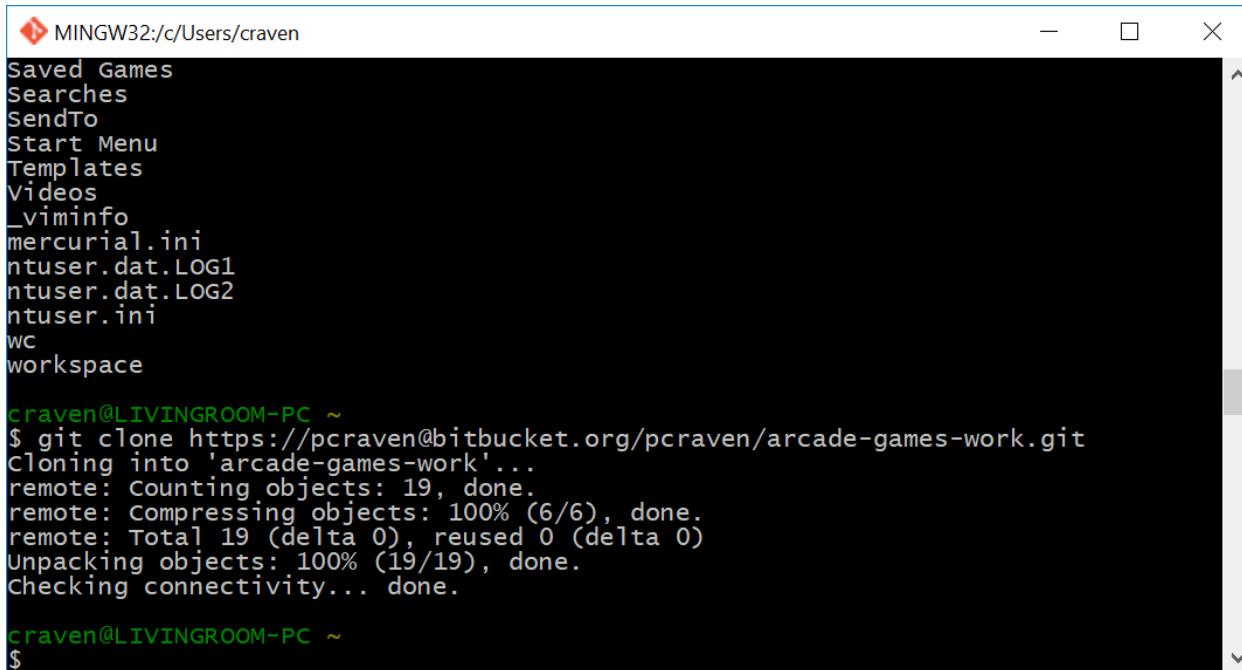
8. Paste the command it gives you in your command prompt:



```
MINGW32:/c/Users/craven
.regtrans-ms
NTUSER.DAT{22422aee-314b-11e7-aba1-a7046d5a93a5}.TMContainer000000000000000000000002
.regtrans-ms
NetHood
OneDrive
Pictures
PrintHood
PycharmProjects
Recent
Saved Games
Searches
SendTo
Start Menu
Templates
Videos
._viminfo
mercurial.ini
ntuser.dat.LOG1
ntuser.dat.LOG2
ntuser.ini
wc
workspace

craven@LIVINGROOM-PC ~
$ git clone https://pcraven@bitbucket.org/pcraven/arcade-games-work.git
```

Then...



```
MINGW32:/c/Users/craven
Saved Games
Searches
SendTo
Start Menu
Templates
Videos
._viminfo
mercurial.ini
ntuser.dat.LOG1
ntuser.dat.LOG2
ntuser.ini
wc
workspace

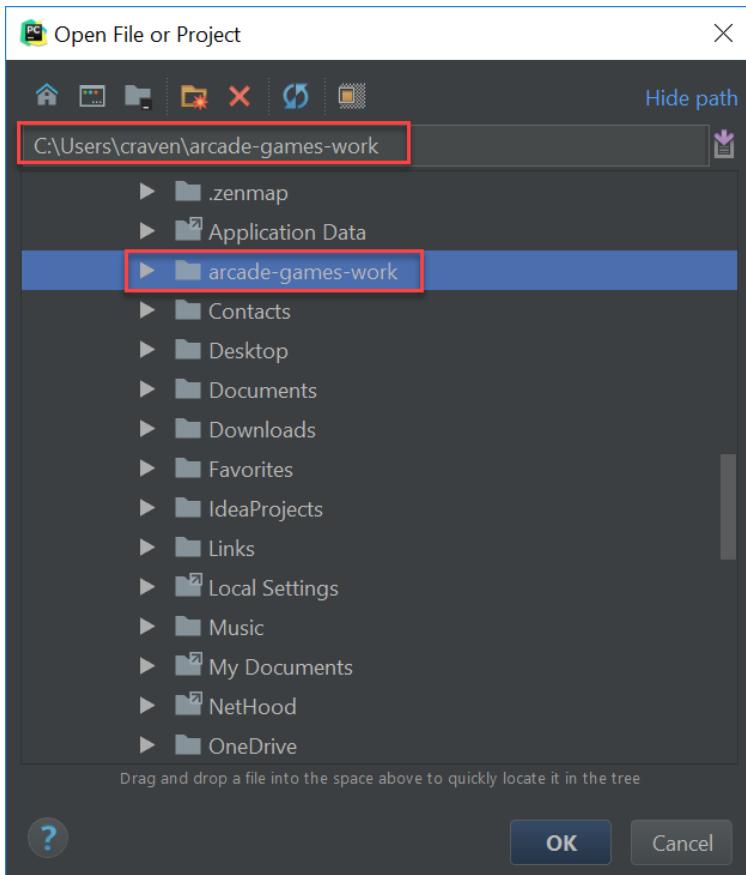
craven@LIVINGROOM-PC ~
$ git clone https://pcraven@bitbucket.org/pcraven/arcade-games-work.git
cloning into 'arcade-games-work'...
remote: Counting objects: 19, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 19 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (19/19), done.
Checking connectivity... done.

craven@LIVINGROOM-PC ~
$
```

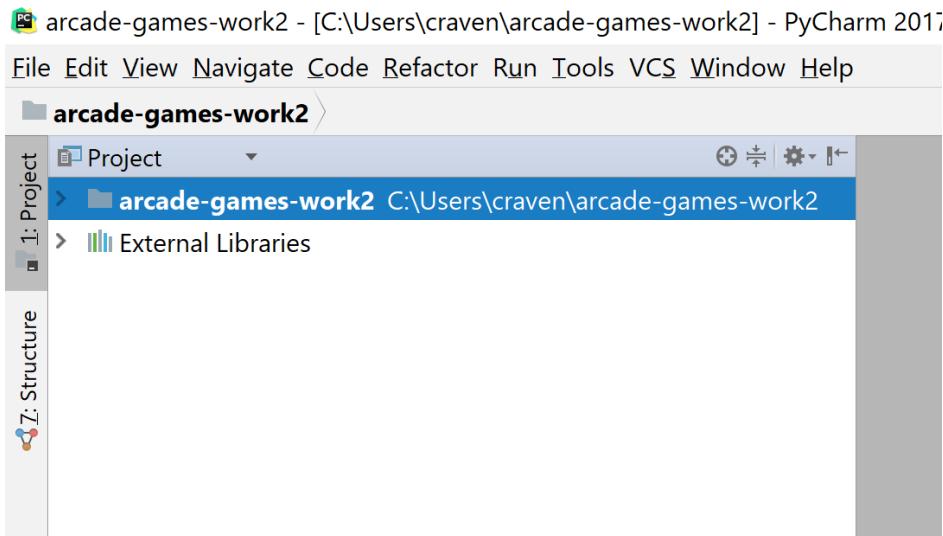
9. There you go! You now have a directory set up where you can do your work.

1.4.7 Open Project in Pycharm

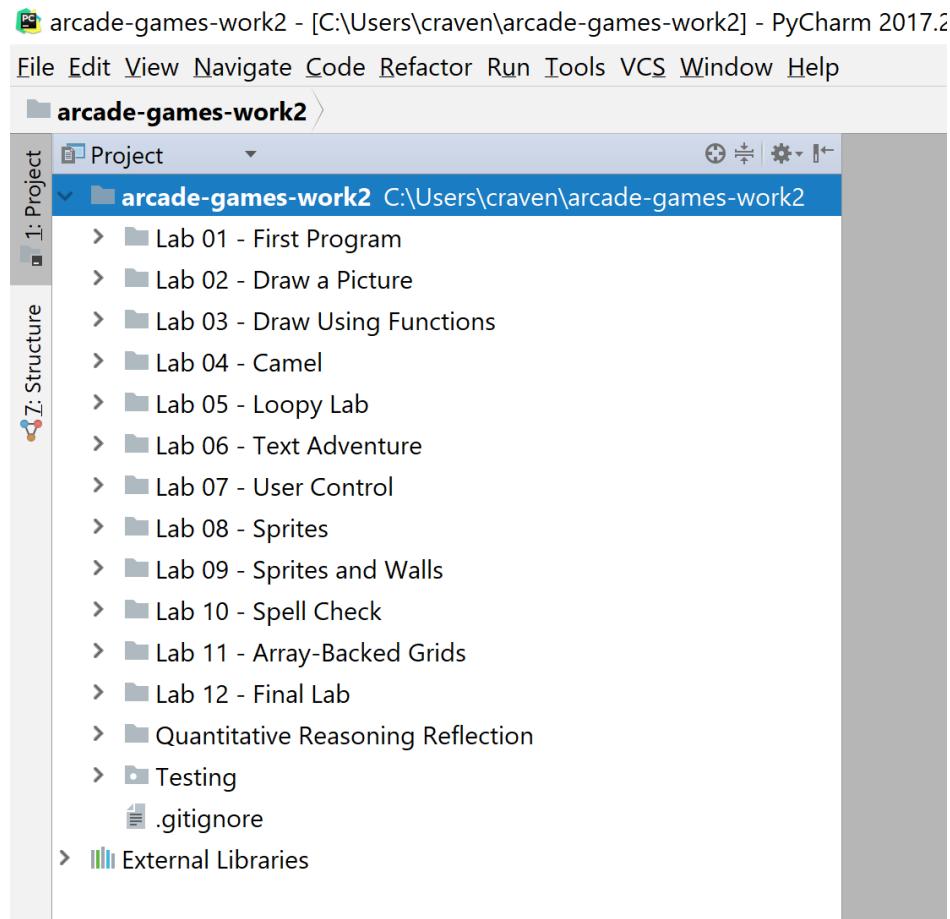
Go ahead and start PyCharm, then select “File... Open” and select that directory.



Your project should look like the image below. If this isn't what you have, you might have opened the wrong folder. Hit "File... Open" and try again.



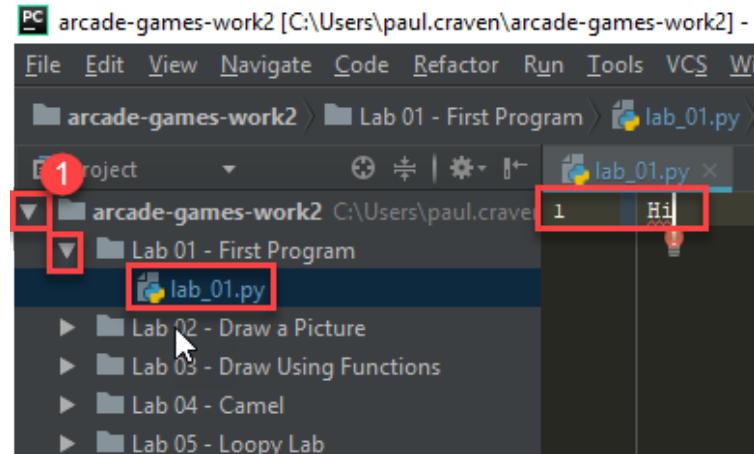
If you click the arrow next to the folder name, you can see all the folders in the project folder.



If you move from computer to computer and have a flash drive, you can reopen your project by just doing “File...Open”. If you don’t have your flash drive, you’ll need to re-clone your repository.

1.4.8 Change a File

Let’s practice making a quick change to one of our files. Open your project folder, open the lab 1 folder, then open lab one. Type in “Hi” or something similar.



Hit Ctrl-S to save.

1.4.9 Commit Your Code

It is time to commit. Wait! You are young and don't want to commit yet?

The cool thing with version control, is that every time you commit, you can go back to the code at that point in time. Version control lets you take it all back! It is the best type of commitment ever!

First, open Git Bash, and switch to the directory with your project using the `cd` command:

```
craven@DESKTOP-RAUFKMA MINGW64 ~
$ cd arcade-games-work2/
```

Optionally, we can use `git status` to see what files have changed:

```
craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   Lab 01 - First Program/lab_01.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Now, add all the files that have changed. The asterisk (*) is a wild card character that means get all changes. Optionally, we could list out each file, but that's a lot of work and we don't want to leave anything behind anyway.

```
craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master)
$ git add *
```

Commit the changes:

```
craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master)
$ git commit -m "Work on lab 1"
[master 45028a5] Work on lab 1
 1 file changed, 1 insertion(+)
```

You might get an error, if the computer doesn't know who you are yet. If you get this error, it will tell you the commands you need to run. They will look like:

```
git config --global user.email "put.your.email.here@my.simpson.edu"
git config --global user.name "Jane Smith"
```

Then you can re-run your commit command. You can use the “up” arrow to get commands you typed in previously so you don't need to retype anything.

1.4.10 Push Your Code

And push them to the server:

```
craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master)
$ git push
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
```

(continues on next page)

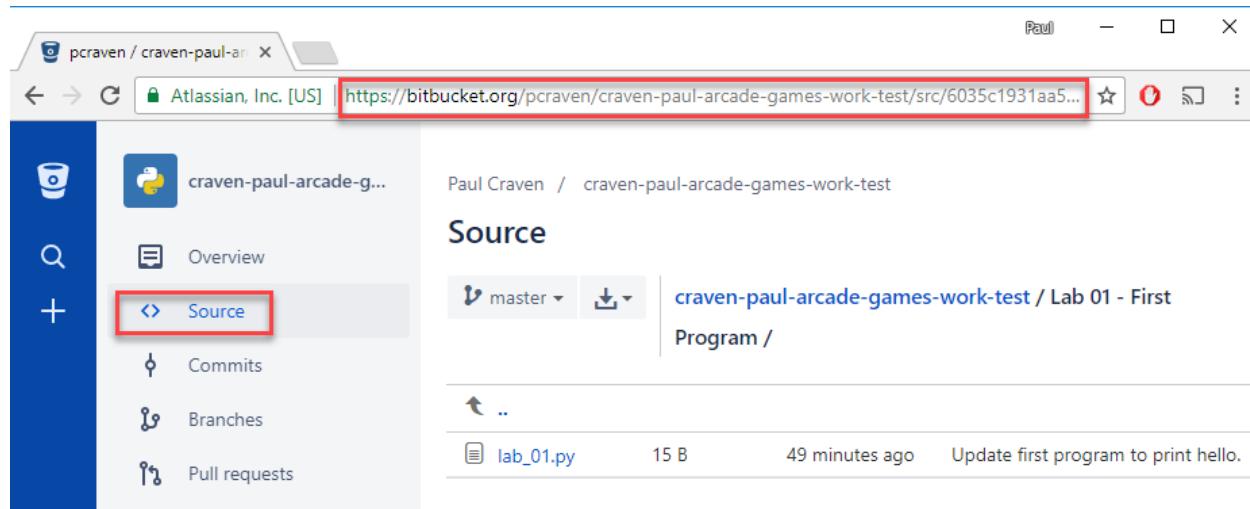
(continued from previous page)

```
Writing objects: 100% (4/4), 329 bytes | 0 bytes/s, done.  
Total 4 (delta 1), reused 0 (delta 0)  
To bitbucket.org:pcraven/arcade-games-work2.git  
 519c361..45028a5 master -> master  
  
craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master)  
$
```

Look to see if the message says that there is an “error.” The message will probably look a little different than what you see above, with other objects or threads, but there should not be any errors. If there are errors, skip down to [What If You Can’t Push?](#).

1.4.11 Turning In Your Programs

When it comes time to turn in one of your programs, go back to BitBucket. Click on “source”, find the lab file, copy the URL:



Now go to Scholar and paste the link into the text field for the lab you are working on.

1.4.12 What If You Can’t Push?

What happens if you can’t push to the server? If you get an error like what’s below? (See highlighted lines.)

```
$ git push  
To bitbucket.org:pcraven/arcade-games-work2.git  
! [rejected]          master -> master (fetch first)  
error: failed to push some refs to 'git@bitbucket.org:pcraven/arcade-games-work2.git'  
'  
hint: Updates were rejected because the remote contains work that you do  
hint: not have locally. This is usually caused by another repository pushing  
hint: to the same ref. You may want to first integrate the remote changes  
hint: (e.g., 'git pull ...') before pushing again.  
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Step 1: Make Sure You Have No Pending Changes

Run a `git status` and make sure you have nothing to commit. It should look like this:

```
craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master)
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working tree clean
```

If you do have code to commit, jump up to [Commit Your Code](#) and then come back here.

Step 2: Pull Changes From The Server

Pull changes from the server:

```
$ git pull
```

Normally, this will work fine and you'll be done. But if you have other computers that you are coding on, the computer will automatically try to merge.

Step 2A: Merging

If you get a screen like the image below, the computer automatically merged your code bases. It now wants you to type in a comment for the merge. We'll take the default comment. Hold down the shift key and type ZZ. If that doesn't work, hit escape, and then try again.

(You are in an editor called **vim** and it is asking you for a comment about merging the files. Unfortunately vim is really hard to learn. Shift-ZZ is the command to save, and all we want to do is get out of it and move on.)

```
Merge branch 'master' of bitbucket.org:pcraven/arcade-games-work2
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
~
```

It should finish with something that looks like:

```
craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master)
Merge made by the 'recursive' strategy.
Lab 01 - First Program/lab_01.py | 3 +--
1 file changed, 2 insertions(+), 1 deletion(-)
```

If instead you get this:

Then we edited the same file in the same spot. We have to tell the computer if we want our changes, or the changes on the other computer.

Step 2B: Resolving a Merge Conflict

Do a git status. It should look something like this:

```
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
(use "git pull" to merge the remote branch into yours)
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:  Lab 01 - First Program/lab_01.py

no changes added to commit (use "git add" and/or "git commit -a")
```

The key thing to look for is any file that says both modified.

If you want **your** copy, type:

```
$ git checkout --ours "Lab 01 - First Program/lab_01.py"
```

If instead you want **their** copy (or the copy on the other computer) type

```
$ git checkout --theirs "Lab 01 - First Program/lab_01.py"
```

Then when you are all done with all merges, type:

```
craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master|MERGING)
$ git add *

craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master|MERGING)
$ git commit -m"Merged"
[master e083f36] Merged

craven@DESKTOP-RAUFKMA MINGW64 ~/arcade-games-work2 (master)
$ git push
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 531 bytes | 0 bytes/s, done.
Total 5 (delta 2), reused 0 (delta 0)
```

(continues on next page)

(continued from previous page)

```
To bitbucket.org:pcraven/arcade-games-work2.git
  6a8f398..e083f36  master -> master
```

Step 3: Try Pushing Again

```
$ git push
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 604 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To bitbucket.org:pcraven/arcade-games-work2.git
  d66b008..aeb9cf3  master -> master
```

1.4.13 Longer Git Command Reference

In my experience with 300 level group-project classes, these commands seem to capture most of what students need to do.

Command	Description
git status	See what has changed
git fetch	Grab stuff from the server, but don't merge
git merge --no-commit --no-ff test_branch	Merge
git merge --abort	Abort a merge
git pull	Fetch and Merge
git add myfile.txt	Add myfile.txt to be committed
git add .	Add everything
grep -r "<< HEAD" *	Search all files to see if there is merge error text. Do this before committing
git checkout --ours "myfile.txt"	Toss your changes in a merge, use theirs
git checkout --theirs "myfile.txt"	Toss their changes, use yours
git checkout -- .	Remove all your changes, go back to what was last committed. Untracked files are kept.
git -f clean	Remove untracked files
git checkout 44fd	Find the hash of a check-in, and you can go back to that check in. (Don't use 44fd, but replace with the hash you want.)
git checkout master	Go back to most recent check in on the master branch.
git commit -m "My message"	Commit your work. Use a descriptive message or the other people in the class will be irritated with you.
git push	Push commit up to the server.

1.5 The `print` Function

1.5.1 Printing Hello World

We will use a function called `print` to print to the screen. `print` is called a *function*.

You've already used functions in mathematics. For example, `sin` and `cos`. Functions are followed by parentheses: `()`. We put the function *parameter(s)* inside the parenthesis.

With a sine function, we put in an angle. With the `print` function, we are going to put the text we want to print. Text must be enclosed in quotes.

```
print("Hello there")
```

Note that case matters. The following will not work:

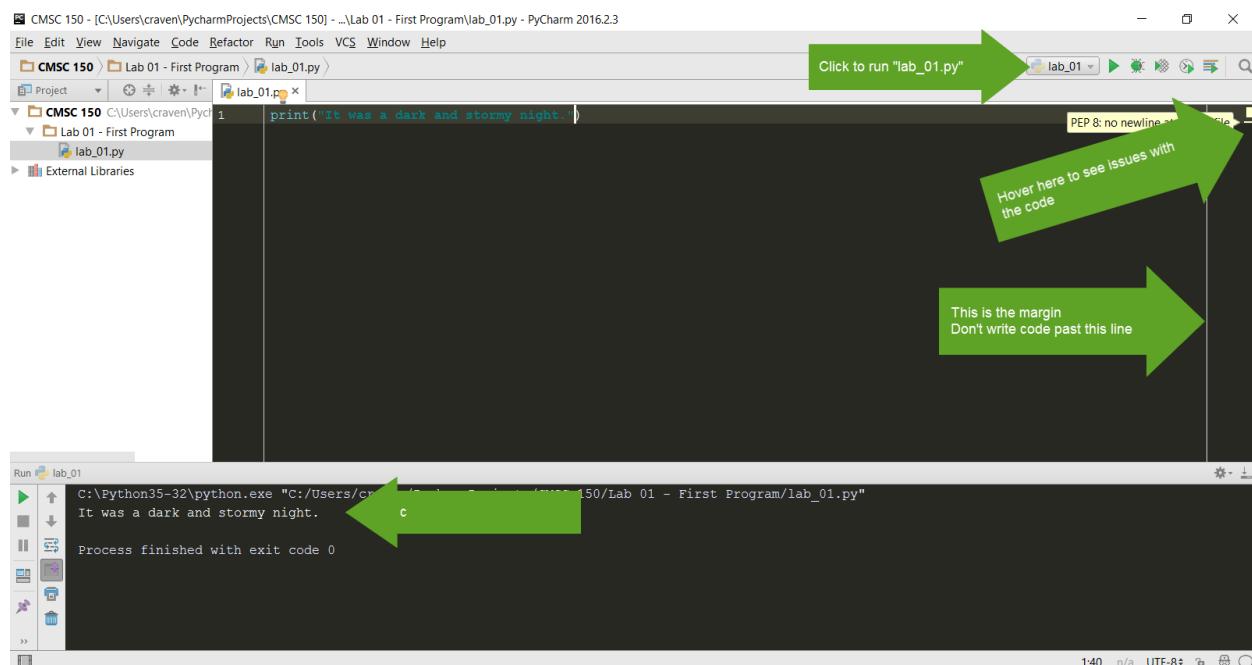
```
Print("Hello there")
```

Great! Time to run it.

Right-click on the program and select “Run ‘lab_01.py’”

Before we go on, note how the PyCharm window is put together. See the output of your program at the bottom of the screen. Click the image below to make it bigger and note the:

- Right margin. You can write code past this point, but don’t.
- Where you can hover your mouse for “hints” on how to make your code better.
- Where you can quickly click to run your program again.



Ok, now it's time to update our program. Go back to our program and improve it by printing multiple lines, while quoting Snoopy's famous story:

1.5.2 Multiple Print Lines

Let's add additional code:

```
print("It was a dark and stormy night.")
print("Suddenly a shot rang out!")
```

Go ahead and run it to make sure it outputs as expected.

1.5.3 Escape Codes

If quotes are used to tell the computer the start and end of the string of text you wish to print, how does a program print out a set of double quotes? (This is a double quote " and this is a single quote ').) For example:

```
print("I want to print a double quote " for some reason.")
```

This code doesn't work. The computer looks at the quote in the middle of the string and thinks that is the end of the text. Then it has no idea what to do with the commands for some reason and the quote and the end of the string confuses the computer even further.

It is necessary to tell the computer that we want to treat that middle double quote as text, not as a quote ending the string. This is easy, just prepend a backslash in front of quotes to tell the computer it is part of a string, not a character that terminates a string. For example:

```
print("I want to print a double quote \" for some reason.")
```

This combination of the two characters \" is called an *escape code*. Almost every language has escape codes. Here's another example:

```
print("Audrey Hepburn once said \"Nothing is impossible. The word itself says 'I'm
↳Possible!' .\"")
```

This will print:

```
Audrey Hepburn once said "Nothing is impossible. The word itself says 'I'm Possible!' .
↳"
```

Because the backslash is used as part of an escape code, the backslash itself must be escaped if you want to use one. For example, this code does not work correctly:

```
print("The file is stored in C:\\new folder")
```

Why? Because \n is an escape code. To print the backslash it is necessary to escape it like so:

```
print("The file is stored in C:\\\\new folder")
```

There are a few other important escape codes to know. Here is a table of the important escape codes:

Escape code	Description
\'	Single Quote
\"	Double Quote
\t	Tab
\r	CR: Carriage Return (move to the left)
\n	LF: Linefeed (move down)

What is a “Carriage Return” and a “Linefeed”? Try this example:

```
print("This\nis\nmy\nsample.")
```

The output from this command is:

```
This  
is  
my  
sample.
```

The \n is a linefeed. It moves “cursor” where the computer will print text down one line. The computer stores all text in one big long line. It knows to display the text on different lines because of the placement of \n characters.

To make matters more complex, different operating systems have different standards on what makes a line ending.

Escape code	Description
\r\n	CR+LF: Microsoft Windows
\n	LF: UNIX based systems, and newer Macs.
\r	CR: Older Mac based systems

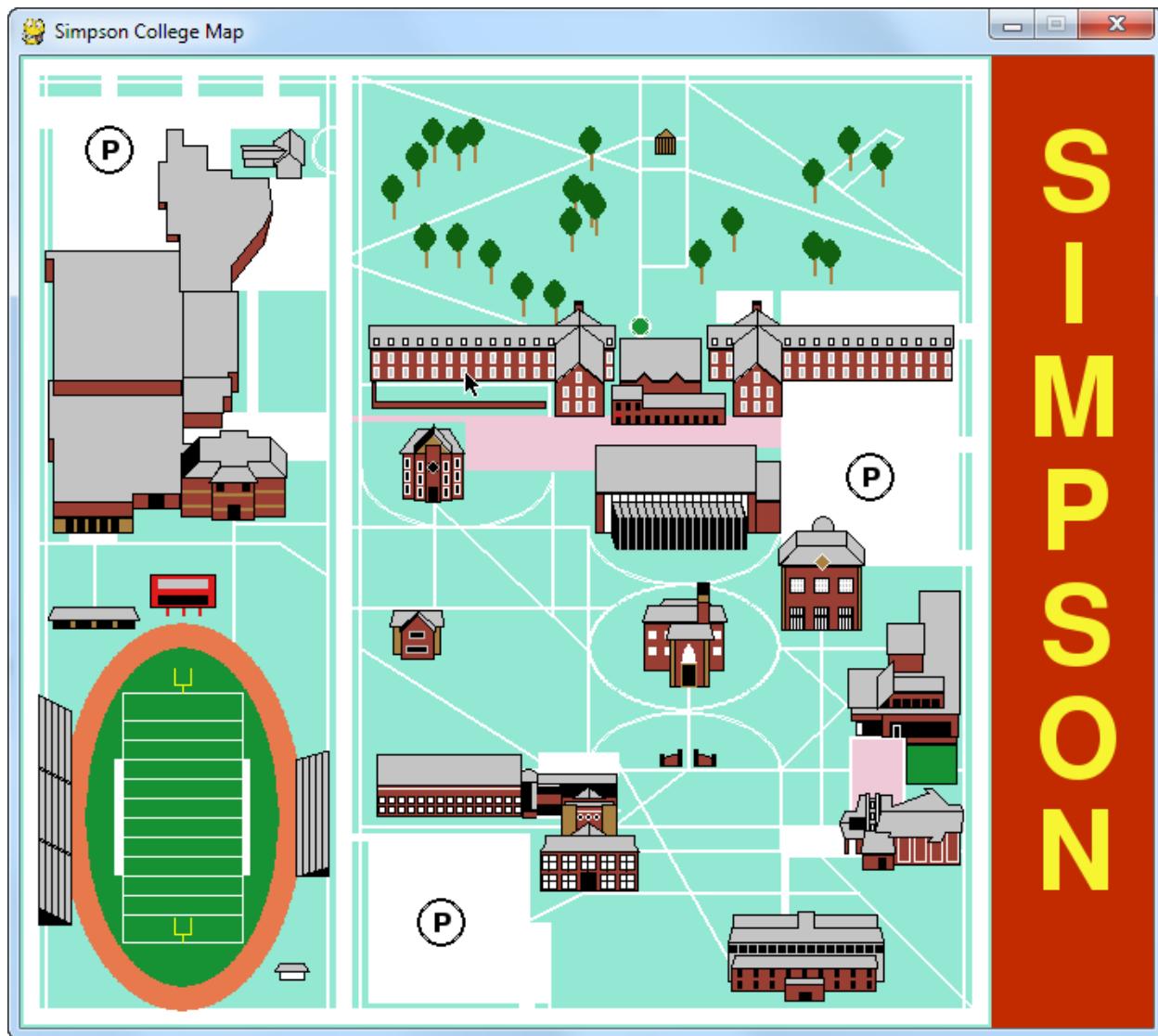
Ok, now it is time to make this lab yours. Write program that consists of several print statements. Here is my example:

```
print("You can print a statement surrounded by double quotes.")  
print('You can print a statement surrounded by single quotes.')  
  
print("If you want to print a double quote, you can by prepending it with")  
print("a slash. \"That's great!\" he said.")  
  
print("If you want to print a backslash, you can by prepending it with")  
print("a slash. So this \\ prints one backslash, and this \\\\\\ does two.")  
  
print("You can print a blank line with a empty print statement.")  
print()  
  
print("You can use a backslash n to print a new line. These\\nare\\non\\nnew\\nlines.")  
print("""You can print  
on multiple  
lines using  
triple  
quotes. Just in  
case you wanted to.""")
```

1.6 How to Draw with Your Computer

Finally, time to do start making graphics!

By the end of this chapter, you should know how to write programs that will draw images on the screen. Below is an example of what one student did:

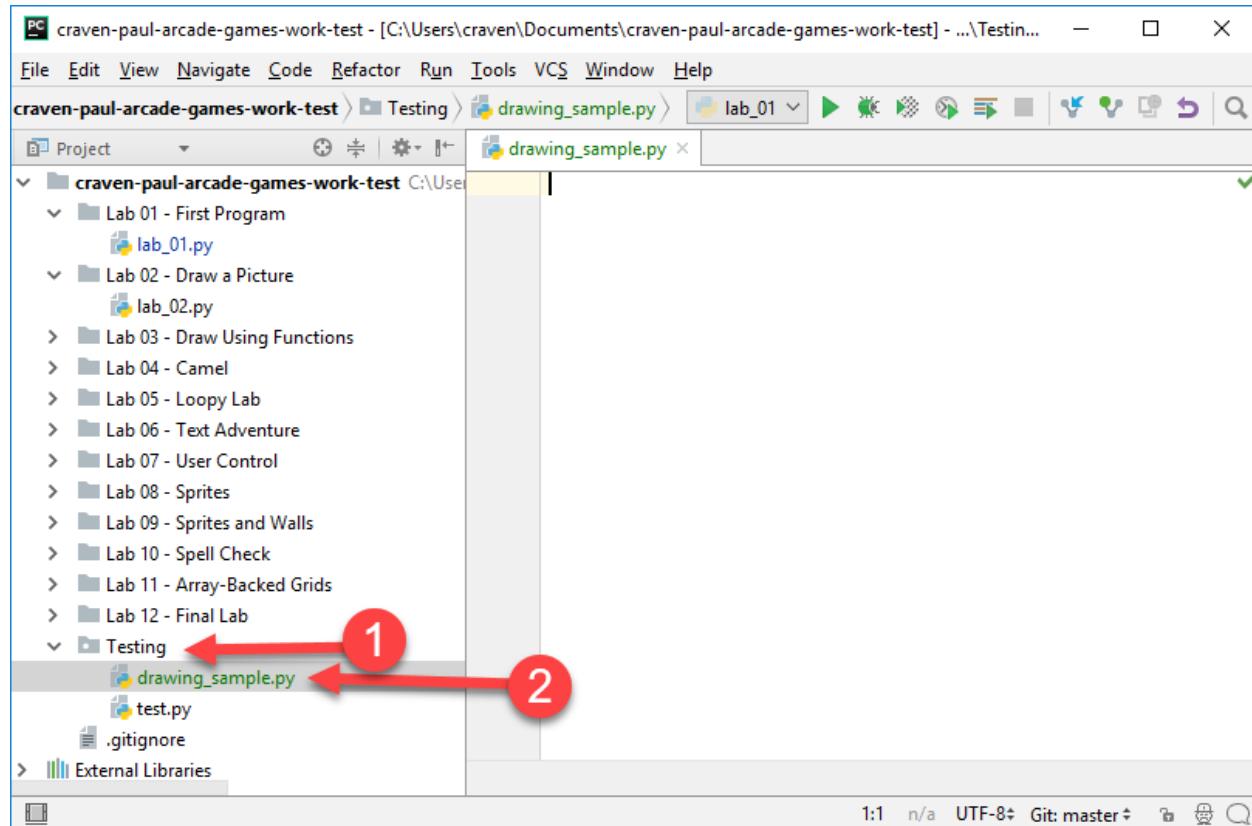


At the end of *Lab 2: Draw a Picture* you can page through several other examples of what students have created for this lab.

1.6.1 Creating a New Program

Open up PyCharm to the same project we created earlier. We'll use it for all our work this semester. Do **not** create new projects for each lab or program you create in this class. Just use one project for the entire class.

We are going to create a lot of code samples as we go through the chapters. Let's create a directory in our project for these samples, away from the labs. Call it `Code Samples`. Then create a Python source file for this chapter called `drawing_samples.py`. Your system should look like this:



1.6.2 Comments

Before we begin learning to draw, we need to learn about **comments** in code. When typing in computer code, sometimes we want to be able to write things that the computer ignores. We call this “commenting our code.” You will see a lot of comments in my code examples to explain how they work.

Below are two ways of adding comments to code in the Python computer language:

```

1 """
2 This is a sample program to show how to draw using the Python programming
3 language and the Arcade library.
4
5 Multi-line comments are surrounded by three of the double-quote marks.
6 Single-line comments start with a hash/pound sign. #
7 """
8
9 # This is a single-line comment.

```

Go ahead and try it. Multi-line comments usually start each source file and explain what the code does.

Let’s try running the program. But before we run the program, we need to make sure we are running the *right* program. Look at the image below. If I select “run” with the green arrow, I will run `lab_01.py`, *not* the program I want to run. You need to right-click on our program and select “Run `drawing_sample.py`” instead.

```

1 """
2 This is a sample program to show how to draw using the Python programming
3 language and the Arcade library.
4
5 Multi-Line comments are surrounded by three of the double-quote marks.
6 Single-Line comments start with a hash/pound sign. #
7 """
8
9 #. This is a single-line comment.
10

```

Run lab_01

```

C:\Program Files (x86)\Python36\python.exe" "C:/Users/craven/Documents/craven-paul-arcade-games-work-test/Lab_01 - First Program
It was a dark and stormy night.
Suddenly a shot rang out!
Process finished with exit code 0

```

Hey wait! When we finally run our program, nothing happens. That's because the only code that we wrote were “comments.” Comments are ignored. Therefore, there was nothing for the computer to do. Read on.

1.6.3 Import the Arcade Library

Before we can draw anything, we need to import a “library” of code that has commands for drawing.

Computer languages come with a set of built-in commands. Most programs will require *more* commands than what the computer language loads by default. These sets of commands are called **libraries**. Some languages have their own special term for these libraries. In the case of Python, they are called **modules**.

Thankfully, it is easy to import a library of code. If we want to use the “arcade” library, all we need to do is add `import arcade` at the top of our program.

Attention: Libraries should always be imported at the **top** of your program. Only comments should appear ahead of an `import` statement. Technically, you *can* put the `import` statement somewhere else, just like technically you *can* wear a pair of shorts on your head. But don’t. Trust me on this.

In the code below, we’ve imported the arcade library. If you run the code, yet again nothing will happen. We’ve asked to load the arcade library, but we haven’t *done* anything with it yet. That’s our next step.

```

1 """
2 This is a sample program to show how to draw using the Python programming
3 language and the Arcade library.
4 """
5

```

(continues on next page)

(continued from previous page)

```
6 # Import the "arcade" library
7 import arcade
```

1.6.4 How to Open a Window for Drawing

Now it is time to open the window. See the command below:

How does this command work? To begin, we select the arcade library with `arcade`. Then we separate the library from the command we want to call with a period: `.`. Next, we put in the name of the command to run. Which happens to be `open_window`.

Note: Commands that we can run are called **functions**.

Just like the sine and cosine functions in math, we surround the function **parameters** with parenthesis. For example:

```
my_function(parameters)
```

The data we need to pass the function are the parameters. In the case of `open_window`, we need three parameters:

- The window width in pixels.
- The window height in pixels.
- The text that will appear on the title bar.

In the case of width and height, the numbers specify the part of the window you can draw on. The actual window is larger to accommodate the title bar and borders. So a 600x600 window is really 602x632 if you count the title bar and borders.

Wait, how do we know that it was the `open_window` function to call? How did we know what parameters to use? The names of the functions, the order of the parameters, is the **Application Program Interface** or “API” for short. You can click here for the [Arcade API](#). Any decent code library will have an API and documentation you can find on the web.

Below is an example program that will open up a window:

```
1 """
2 This is a sample program to show how to draw using the Python programming
3 language and the Arcade library.
4 """
5
6 # Import the "arcade" library
7 import arcade
8
9 # Open up a window.
10 # From the "arcade" library, use a function called "open_window"
11 # Set the and dimensions (width and height)
12 # Set the window title to "Drawing Example"
13 arcade.open_window(600, 600, "Drawing Example")
```

Try running the code above. It kind-of works. If you have fast eyes, and a slow computer you might see the window pop open, then immediately close. If your computer is fast, you won’t see anything at all because the window closes too fast. Why does it close? Because our program is done! We’ve ran out of code to execute.

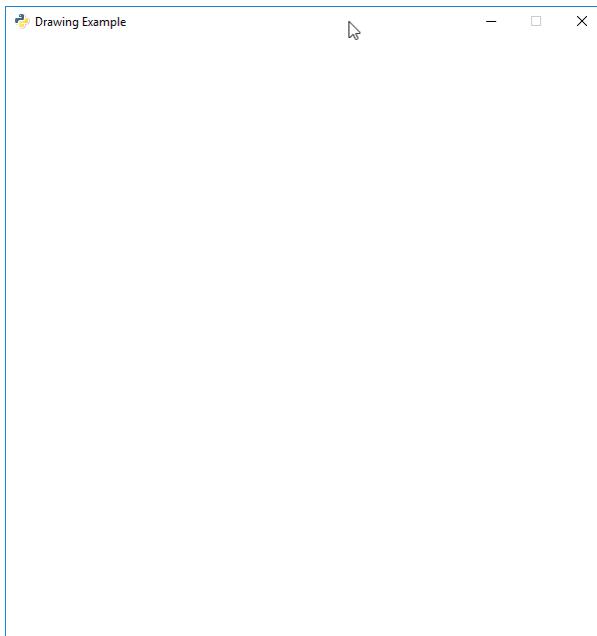
To keep the window open, we need to pause until the user hits the close button. To do this, we'll use the `run` command in the Arcade library. The `run` command takes no parameters, but even if a function doesn't take parameters, you still need to use parenthesis.

```

1  """
2  This is a sample program to show how to draw using the Python programming
3  language and the Arcade library.
4  """
5
6  # Import the "arcade" library
7  import arcade
8
9  # Open up a window.
10 # From the "arcade" library, use a function called "open_window"
11 # Set the window title to "Drawing Example"
12 # Set the and dimensions (width and height)
13 arcade.open_window(600, 600, "Drawing Example")
14
15 # Keep the window up until someone closes it.
16 arcade.run()

```

You should get a window that looks something like this:



1.6.5 Clearing the screen

Right now we just have a default white as our background. How do we get a different color? Use the `set_background_color` command.

But by itself, the function doesn't work. You need a two more commands. These tell the Arcade library when you are about to start drawing (`start_render`), and when you are done drawing (`finish_render`).

See below:

```

1  """
2  This is a sample program to show how to draw using the Python programming

```

(continues on next page)

(continued from previous page)

```
3 language and the Arcade library.  
4 """  
5  
6 # Import the "arcade" library  
7 import arcade  
8  
9 # Open up a window.  
10 # From the "arcade" library, use a function called "open_window"  
11 # Set the window title to "Drawing Example"  
12 # Set the and dimensions (width and height)  
13 arcade.open_window( 600, 600, "Drawing Example")  
14  
15 # Set the background color  
16 arcade.set_background_color(arcade.color.AIR_SUPERIORITY_BLUE)  
17  
18 # Get ready to draw  
19 arcade.start_render()  
20  
21 # (The drawing code will go here.)  
22  
23 # Finish drawing  
24 arcade.finish_render()  
25  
26 # Keep the window up until someone closes it.  
27 arcade.run()
```

1.6.6 Specifying Colors

Wait, where did AIR_SUPERIORITY_BLUE come from? How do I get to choose the color I want? There are two ways to specify colors:

- Look at the `arcade.color` API documentation and specify by name.
- Specify the RGB or RGBA color.

To specify colors by name, you can look at the color API documentation and use something like `arcade.color.AQUAMARINE` in your program. Then color names come from the ColorPicker [color chart](#).

If the color you want isn't in the chart, or you just don't want to use that chart, you can specify colors by "RGB". RGB stands for Red, Green, and Blue.

Computers, TVs, color changing LEDs, all work by having three small lights close together. A red light, a green light, and a blue light. Turn all three lights off and you get black. Turn all three lights on and you get white. Just turn on the red, and you get red. Turn on both red and green to get yellow.

RGB based monitors work on an *additive* process. You start with black and add light to get color.

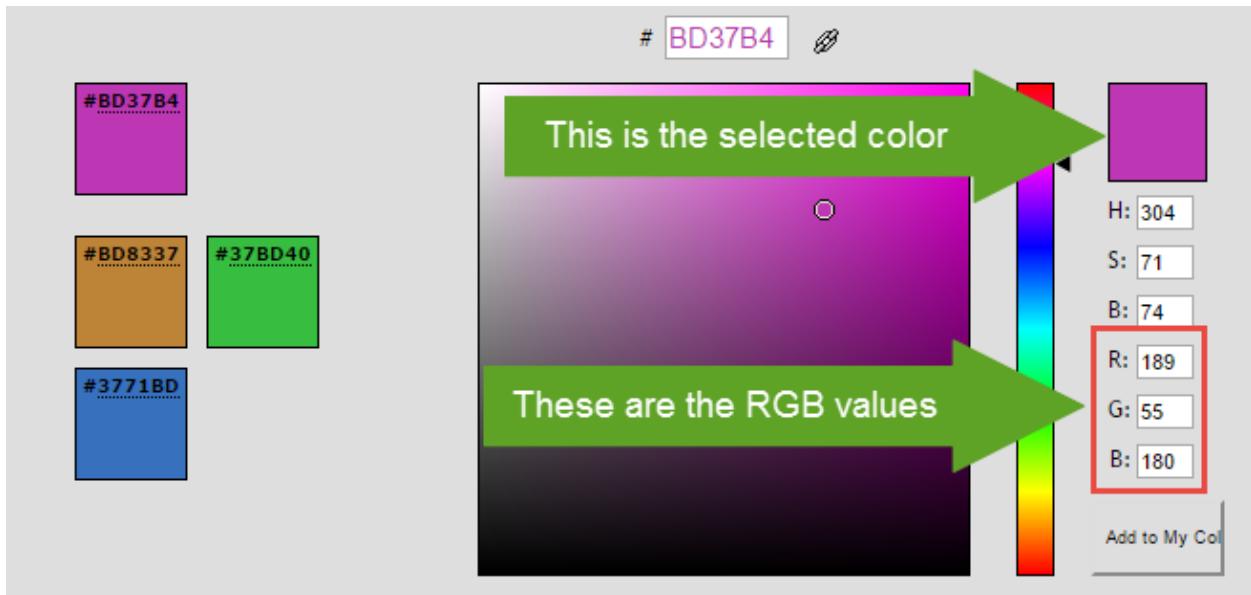
This is different than paint or ink, which works on a *subtractive* process. You start with white and add to get darker colors.

Therefore, keep separate in your mind how light-based RGB color works from how paint and ink works.

We specify how much red, green, and blue to use using numbers. No light is zero. Turn the light on all the way and it is 255. So `(0, 0, 0)` means no red, no green, no blue. Black. Here are some other examples:

Red	Green	Blue	Color
0	0	0	Black
255	255	255	White
127	127	127	Gray
255	0	0	Red
0	255	0	Green
0	0	255	Blue
255	255	0	Yellow

There are tools that let you easily find a color, and then get the RGB values. One I really like, because it is easy to remember is “colorpicker.com”. You can select the color, and then get the numbers to use when specifying a color. See the image below:



After getting the number, specify the color as a set of three numbers surrounded by parenthesis, like this:

```
arcade.set_background_color((189, 55, 180))
```

In addition to RGB, you can also specify “Alpha.” The “Alpha Channel” controls how transparent the color is. If you draw a square with an alpha of 255, it will be solid and hide everything behind it. An alpha of 127 will be in the middle, you will see some of the items behind the square. An alpha of 0 is completely transparent and you’ll see nothing of the square.

Wait, What Is Up With 255?

Notice how the color values go between 0 and 255? That’s strange. Why 255? Why not 100? Why not 1000?

The reason is important to understand how computers work. Remember how everything is stored in numbers? They are not just stored in numbers, they are stored in 1’s and 0’s.

Everything to the computer is a switch. If there is electricity, we have a 1. If there is no electricity we have a 0. We can store those 1’s and 0’s in memory. We call these 1’s and 0’s **binary numbers**.

How do we go from 1’s and 0’s to numbers we normally use? For example, a number like 758? We do that with a combination of 1’s and 0’s. Like this:

Binary - Base 2	Base 10
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8

See the pattern? It is the same pattern we use when we count as a kid. As a kid we learned to go 0 to 9, then when we hit 9 we go back to 0 and add one to the ten's place. Here we only have 0 to 1 instead of 0 to 9. And instead of a “ten's place” we have a “two's place.”

You might have used “bases” in math class long ago. Computers work in Base-2 because they only have two ways to count, on or off. Humans think in Base-10 because we have 10 fingers.

Numbers are stored in **bytes**. A byte is a set of eight binary numbers. If we were to follow the pattern we started above, the largest number we could store with eight 1's and 0's is:

1111 1111

In Base-10 this is 255.

Let's use some math. We have 8 ones and zeros. That give us $2^8 = 256$ possible numbers. Since zero is a combination, that makes the biggest number 255.

If we had 16 bits, then we'd have $2^{16} = 65,536$ possible combinations. Or a number from 0-65535. A 32-bit computer can hold numbers up to $2^{32} = 4,294,967,296$. A 64-bit computer can hold really large numbers!

So because a computer holds colors with one byte for red, one for green, and one for blue, each color has a value range from 0 - 255.

1.6.7 The Coordinate System

In your math classes, you've learned about the Cartesian coordinate system, which looks like this:

Fig. 6: Source: Wikipedia: Cartesian coordinate system

Our graphics will be drawn using this same system. But there are additional things to keep in mind:

- We will only draw in the upper right quadrant. So 0,0 will be in the lower left of the screen, and all negative coordinates will be off-screen.
- Each “Point” will be a pixel. So a window that is 800 pixels wide, will have x-coordinates that run from 0 to 800.

1.6.8 Drawing a Rectangle

Let's start drawing with a program to draw a rectangle. The function we will use is `draw_lrtb_rectangle_filled`. It stands for “draw left-right-top-bottom rectangle”.

We'll use this program to draw a green rectangle:

```

1 """
2 This is a sample program to show how to draw using the Python programming
3 language and the Arcade library.
4 """
5
6 # Import the "arcade" library
7 import arcade
8
9 # Open up a window.
10 # From the "arcade" library, use a function called "open_window"
11 # Set the window title to "Drawing Example"
12 # Set the and dimensions (width and height)
13 arcade.open_window(600, 600, "Drawing Example")
14
15 # Set the background color
16 arcade.set_background_color(arcade.color.AIR_SUPERIORITY_BLUE)
17
18 # Get ready to draw
19 arcade.start_render()
20
21 # Draw a rectangle
22 # Left of 5, right of 35
23 # Top of 590, bottom of 570
24 arcade.draw_lrtb_rectangle_filled(5, 35, 590, 570, arcade.color.BITTER_LIME)
25
26 # Finish drawing
27 arcade.finish_render()
28
29 # Keep the window up until someone closes it.
30 arcade.run()

```

There are a lot of shapes we can draw. Try running the program below:

```

1 """
2 This is a sample program to show how to draw using the Python programming
3 language and the Arcade library.
4 """
5
6 # Import the "arcade" library
7 import arcade
8
9 # Open up a window.
10 # From the "arcade" library, use a function called "open_window"
11 # Set the window title to "Drawing Example"
12 # Set the and dimensions (width and height)
13 arcade.open_window(600, 600, "Drawing Example")
14
15 # Set the background color
16 arcade.set_background_color(arcade.color.AIR_SUPERIORITY_BLUE)
17
18 # Get ready to draw
19 arcade.start_render()
20
21 # Draw a rectangle
22 # Left of 5, right of 35
23 # Top of 590, bottom of 570
24 arcade.draw_lrtb_rectangle_filled(5, 35, 590, 570, arcade.color.BITTER_LIME)

```

(continues on next page)

(continued from previous page)

```
25
26 # Different way to draw a rectangle
27 # Center rectangle at (100, 520) with a width of 45 and height of 25
28 arcade.draw_rectangle_filled(100, 520, 45, 25, arcade.color.BLUSH)
29
30 # Rotate a rectangle
31 # Center rectangle at (200, 520) with a width of 45 and height of 25
32 # Also, rotate it 45 degrees.
33 arcade.draw_rectangle_filled(200, 520, 45, 25, arcade.color.BLUSH, 45)
34
35
36 # Draw a point at (50, 580) that is 5 pixels large
37 arcade.draw_point(50, 580, arcade.color.RED, 5)
38
39 # Draw a line
40 # Start point of (75, 590)
41 # End point of (95, 570)
42 arcade.draw_line(75, 590, 95, 570, arcade.color.BLACK, 2)
43
44 # Draw a circle outline centered at (140, 580) with a radius of 18 and a line
45 # width of 3.
46 arcade.draw_circle_outline(140, 580, 18, arcade.color.WISTERIA, 3)
47
48 # Draw a circle centered at (190, 580) with a radius of 18
49 arcade.draw_circle_filled(190, 580, 18, arcade.color.WISTERIA)
50
51 # Draw an ellipse. Center it at (240, 580) with a width of 30 and
52 # height of 15.
53 arcade.draw_ellipse_filled(240, 580, 30, 15, arcade.color.AMBER)
54
55 # Draw text starting at (10, 450) with a size of 20 points.
56 arcade.draw_text("Simpson College", 10, 450, arcade.color.BRICK_RED, 20)
57
58 # Finish drawing
59 arcade.finish_render()
60
61 # Keep the window up until someone closes it.
62 arcade.run()
```

1.6.9 Drawing Primitives

For a program showing all the drawing primitives, see the example [Drawing Primitives](#). Also, see the API documentation's [Quick Index](#).

What's next? Try [Lab 2: Draw a Picture](#).

1.7 Variables and Expressions

We've learned how to call functions to draw shapes like circles, but what if we could expand that? What if we could create our own instructions for drawing trees, houses, spaceships, and rainbows? What if our program could look like:

```
draw_tree(225, 35)
draw_tree(420, 45)
```

(continues on next page)

(continued from previous page)

```
draw_house(720, 60)
draw_snow_person(300, 20)
```

For this level of computer-programming power, we need to learn three things:

- How to use variables (this chapter)
- How to write expressions (this chapter)
- How to create our own functions (next two chapters)

1.7.1 How to Use Variables

A **variable** is a value the computer stores in memory that can change. That is, it *varies*.

You've used variables in mathematics before. With computer science, we use them a lot. But in math class, you were given the equation and you had to solve for the variable. In computer science class, *we* come up with the equation and the *computer* solves the variable.

Here is a quick example:

```
# What will this print?
x = 5
print(x)
```

What will the code above print? It will print 5.

The = is called an **assignment operator**. It assigns the value on the right side to the variable on the left.

Here's another example. Very similar, but something is different. What will it print?

```
# What will this print?
x = 5
print("x")
```

The code above prints x. Why not 5? Because:

- If there are no quotes, the computer evaluates code like a mathematical expression.
- If there are quotes, we treat what is between the quotes as a string of characters and don't change it.

In fact, that is what we call the characters between the quotes. A **string**, which is short for "string of characters." We don't call it "text."

The following code won't print at all:

```
print(Have a great day!)
```

The code above will fail because the computer will think that it should evaluate Have a great day! as a mathematical expression. It isn't, so the computer gets confused and generates an error. That's why we need quotes:

```
print("Have a great day!")
```

Variable and Function Names

Variable names and function names follow the same rules. There are names you *should* use, names you *shouldn't* use, and names you *can't* use.

Good variable name examples:

- temperature_in_celsius
- tree_position
- car_speed
- number_of_children
- simpson

Legal, but bad variable names:

- temperatueInCelsius - Uses capital letters. Keep it lower case and use underscores.
- x - Too short, and not descriptive.
- Simpson - Starts with a capital letter.

Variable names that won't work:

- tree position - Can't use spaces
- 4runner - Can't start with a number

Sometimes we want to create a variable that won't change. We call these variables **constants**. By convention, these variable names are in all upper case. They are the only variables that use upper-case. For example:

```
PI = 3.14159
SCREEN_WIDTH = 600
RED = (255, 0 ,0)
```

Good variable names help make code *readable*. Note the example below that calculates miles-per-gallon. It isn't easy to understand.

```
# Calculate mpg using confusing variable names
m = 294
g = 10.5
m2 = m / g
print(m2)
```

But the code below that uses descriptive variable names *is* easy to understand.

```
# Calculate mpg using good variable names
miles_driven = 294
gallons_used = 10.5
mpg = miles_driven / gallons_used
print(mpg)
```

1.7.2 How to Create Expressions

Using Operators in Expressions

Great! We are part-way there. To really be powerful, variables need to be used with **expressions**. An expression is simply a mathematical equation like what you've used in math before. Here's an example:

```
# What will this print?
x = 5 + 10
print(x)
```

As you can probably guess, this will print out 15. We call the `+` sign an **operator**. Here are some other operators:

Operator	Description
<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>**</code>	Exponentiation (raise to the power)
<code>/</code>	Division
<code>//</code>	Integer division (rounds down)
<code>%</code>	Modulus (gives remainder of division)

There are two things that **don't** work like you'd expect. There is no “juxtaposition” used to multiply items. And the `=` is not an algebraic equality

Juxtaposition Doesn't Work

Juxtaposition doesn't work for multiplication. For example, the following will **not** work:

```
# The last two lines will error
x = 3
y = 2x
z = 2(3 + x)
```

You can rewrite the code above to work by explicitly multiplying:

```
# This code works. Although it doesn't print anything.
x = 3
y = 2 * x
z = 2 * (3 + x)
```

Easy enough, just remember to use `*` any time you want to multiply.

Assignment Operators

The `=` doesn't work the same as in algebra. The `=` evaluates what is on the right, and puts it in the variable on the left. For example:

```
# This works
x = 3 + 4

# This doesn't work because the only thing that can be on the left of
# the = is one variable.
3 + 4 = x

# This works
x = 5
y = 6
z = x + 2 * y

# This doesn't
x = 5
y = 6
2 * z = x + y
```

This allows us to do some strange things we can't do in algebra!

```
# This works, and prints "3"
x = 3
print(x)

# This works too, even if it is invalid in algebra.
# It takes the value of x (which is 3) and adds one. Then stores
# the result (4) back in x. So we'll print "4".
x = x + 1
print(x)
```

The = sign is also considered an operator. Specifically an “assignment operator.” Here are some other “assignment” operators:

Operator	Description
=	Assignment
+=	Increment
-=	Decrement
*=	Multiply/Add

```
# This works, and prints "3"
x = 3
print(x)

# Make x bigger by one
x = x + 1
print(x)

# Make x bigger by one, just like before
x += 1
print(x)

# Make x smaller by five
x -= 5
print(x)
```

Using Expressions In Function Calls

We can use expressions even in the calls that we make. For example, what if we want to draw a circle in the center of the screen?

We could do something like:

```
1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6 arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing Example")
7
8 arcade.set_background_color(arcade.color.WHITE)
9
10 arcade.start_render()
11
12 # Instead of this:
```

(continues on next page)

(continued from previous page)

```

13 # arcade.draw_circle_filled(400, 300, 50, arcade.color.FOREST_GREEN)
14 # do this:
15 arcade.draw_circle_filled(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2, 50, arcade.color.
16   FOREST_GREEN)
17 arcade.finish_render()
18 arcade.run()

```

Order of Operations

Python will evaluate expressions using the same order of operations that are expected in standard mathematical expressions. For example this equation does not correctly calculate the average:

```
average = 90 + 86 + 71 + 100 + 98 / 5
```

The first operation done is $98/5$. The computer calculates:

$$90 + 86 + 71 + 100 + \frac{98}{5}$$

rather than the desired:

$$\frac{90 + 86 + 71 + 100 + 98}{5}$$

By using parentheses this problem can be fixed:

```
average = (90 + 86 + 71 + 100 + 98) / 5
```

1.7.3 Printing Variables

How can you print variables and text together? Say you've got a variable `result` and you want to nicely print it. Based on what we learned so far, you can do this:

```
answer = "bananas"
print(answer)
```

But that just prints out bananas on a line by itself. Not very descriptive. What if we wanted:

```
The answer is bananas
```

You can do this with:

```
answer = "bananas"
print("The answer is", answer)
```

Better. But I want to add punctuation. If we do this:

```
answer = "bananas"
print("The answer is", answer, ".")
```

We get an extra space before the period:

```
The answer is bananas .
```

The , adds a space when we use it in a print statement. We don't always want that. We can instead use a + sign:

```
answer = "bananas"  
print("The answer is" + answer + ".")
```

Which gets rid of all the spaces:

```
The answer isbananas.
```

So we need to add a space INSIDE the quotes where we want it:

```
answer = "bananas"  
print("The answer is " + answer + ".")
```

Ok, so I think I know how to print variables. Until I try this:

```
answer = 42  
print("The answer is " + answer + ".")
```

The computer doesn't know how to put text and numbers together. If you add two *numbers* 20 + 20 you get 40. If you add two *strings* "20" + "20" you get 2020, but the computer has no idea what to do with a combo of text and numbers. So the fix is to use the str function which converts the number to a string (text):

```
answer = 42  
print("The answer is " + str(answer) + ".")
```

Yes, this is a bit complex. But wait! There's more! Another way to print variables is to use a *formatted string*. Later we will spend a whole other chapter on formatted strings, but they look like:

```
answer = 42  
print(f"The answer is {answer}.")
```

Note we start the string with an f before the quote, and the variable we want to print goes in curly braces.

1.8 Creating Functions

1.8.1 Creating Simple Functions

Defining a function is rather easy.

- Start with the keyword def, which is short for “define.”
- Next, give the function a name. There are rules for function names. They must:
 - Start with a lower case letter.
 - After the first letter, only use letters, numbers, and underscores.
 - Spaces are not allowed. Use underscores instead.
 - While upper-case letters can be used, function names are normally all lower-case.
- After that, we have a set of parenthesis. Inside the parenthesis will go **parameters**. We'll explain those in a bit.
- Next, a colon.

- Everything that is part of the function will be indented four spaces.
- Usually we start a function with a multi-line comment that explains what the function does.

Note: Function definitions go *below* the `import` statements, and *above* the rest of the program. While you can put them somewhere else, you shouldn't.

Below is a program that defines and uses the function twice.

```

1 def print_hello():
2     """ This is a comment that describes the function. """
3     print("Hello!")
4
5
6 print_hello()
7 print_hello()
```

You can define and use multiple functions. But all function definitions should go before the main code.

```

1 def print_hello():
2     print("Hello!")
3
4
5 def print_goodbye():
6     print("Bye!")
7
8
9 print_hello()
10 print_goodbye()
```

Actually, almost *all* code should go in a function. It is a good practice to put the main starting point of your program in a function called `main` and call it.

```

1 def print_hello():
2     print("Hello!")
3
4
5 def print_goodbye():
6     print("Bye!")
7
8
9 def main():
10     """ This is my main program function """
11     print_hello()
12     print_goodbye()
13
14
15 # Run the main program
16 main()
```

An even better way of writing this is with a check to make sure we are trying to run this file, and not import it. The statement for this looks a little weird. In fact, it is weird enough I just look it up and copy/paste it any time I want to use it. Don't worry about understanding how it works yet.

```

1 def print_hello():
2     print("Hello!")
```

(continues on next page)

(continued from previous page)

```

3
4
5 def print_goodbye():
6     print("Bye!")
7
8
9 def main():
10    print_hello()
11    print_goodbye()
12
13
14 # Only run the main function if we are running this file. Don't run it
15 # if we are importing this file.
16 if __name__ == "__main__":
17     main()

```

1.8.2 Taking In Data

Functions are even more powerful if we have them take in data.

Here is a simple example that will take in a number and print it. Notice how I've created a new variable `my_number` in between the parenthesis. This variable will be given whatever value is passed in. In the example below, it is given first a 55, then 25, and finally a 5.

```

1 def print_number(my_number):
2     print(my_number)
3
4
5 print_number(55)
6 print_number(25)
7 print_number(8)

```

You can pass in multiple numbers, just separate them with a comma.

```

1 def add_numbers(a, b):
2     print(a + b)
3
4
5 add_numbers(11, 7)

```

Occasionally, new programmers want to set the number values inside the function. This is wrong. Then the function would only work for those values. The power is in specifying the numbers outside the function. We don't want the function to be limited to only certain data values.

```

1 # This is wrong
2 def add_numbers(a, b):
3     a = 11
4     b = 7
5     print(a + b)
6
7
8 add_numbers(11, 7)

```

1.8.3 Returning and Capturing Values

Functions can not only take in values, functions can return values.

Returning values

For example:

Function that returns two numbers added together

```
1 # Add two numbers and return the results
2 def sum_two_numbers(a, b):
3     result = a + b
4     return result
```

Note: Return is not a function, and does not use parentheses. Don't do `return (result)`.

This only gets us half-way there. Because if we call the function now, not much happens. The numbers get added. They get returned to us. But we do nothing with the result.

```
# This doesn't do much, because we don't capture the result
sum_two_numbers(22, 15)
```

Capturing Returned Values

We need to capture the result. We do that by setting a variable equal to the value the function returned:

```
# Capture the function's result into a variable
# by putting "my_result =" in front of it.
# (Use whatever variable name best describes the data,
# don't blindly use 'my_result' for everything.)
my_result = sum_two_numbers(22, 15) # <--- This line CAPTURES the return value

# Now that I captured the result, print it.
print(my_result) # <--- This is printing, NOT capturing.
```

Now the result isn't lost. It is stored in `my_result` which we can print or use some other way.

Volume Cylinder Example

Function that returns the volume of a cylinder

```
1 def volume_cylinder(radius, height):
2     pi = 3.141592653589
3     volume = pi * radius ** 2 * height
4     return volume
```

Because of the return, this function could be used later on as part of an equation to calculate the volume of a six-pack like this:

```
six_pack_volume = volume_cylinder(2.5, 5) * 6
```

The value returned from `volume_cylinder` goes into the equation and is multiplied by six.

There is a big difference between a function that prints a value and a function that returns a value. Look at the code below and try it out.

```
1 # Function that prints the result
2 def sum_print(a, b):
3     result = a + b
4     print(result)
5
6
7 # Function that returns the results
8 def sum_return(a, b):
9     result = a + b
10    return result
11
12
13 # This prints the sum of 4+4
14 sum_print(4, 4)
15
16 # This does not
17 sum_return(4, 4)
18
19 # This will not set x1 to the sum
20 # It actually gets a value of 'None'
21 x1 = sum_print(4, 4)
22 print("x1 =", x1)
23
24 # This will set x2 to the sum
25 # and print it properly
26 x2 = sum_return(4, 4)
27 print("x2 =", x2)
```

When first working with functions it is not unusual to get stuck looking at code like this:

```
def calculate_average(a, b):
    """ Calculate an average of two numbers """
    result = (a + b) / 2
    return result

# Pretend you have some code here
x = 45
y = 56

# Wait, how do I print the result of this?
calculate_average(x, y)
```

How do we print the result of calculate_average? The program can't print result because that variable only exists inside the function. Instead, use a variable to capture the result:

```
def calculate_average(a, b):
    """ Calculate an average of two numbers """
    result = (a + b) / 2
    return result

# Pretend you have some code here
x = 45
y = 56

average = calculate_average(x, y)
print(average)
```

1.8.4 Documenting Functions

Functions in Python typically have a comment as the first statement of the function. This comment is delimited using three double quotes, and is called a docstring. A function may look like:

```
def volume_cylinder(radius, height):
    """Returns volume of a cylinder given radius, height."""
    pi = 3.141592653589
    volume = pi * radius ** 2 * height
    return volume
```

The great thing about using docstrings in functions is that the comment can be pulled out and put into a website documenting your code using a tool like Sphinx. Most languages have similar tools that can help make documenting your code a breeze. This can save a lot of time as you start working on larger programs.

1.8.5 Variable Scope

The use of functions introduces the concept of scope. Scope is where in the code a variable is “alive” and can be accessed. For example, look at the code below:

```
# Define a simple function that sets
# x equal to 22
def f():
    x = 22

# Call the function
f()
# This fails, x only exists in f()
print(x)
```

The last line will generate an error because x only exists inside of the f() function. The variable is created when f() is called and the memory it uses is freed as soon as f() finishes.

Here’s where it gets complicated. A more confusing rule is accessing variables created outside of the f() function. In the following code, x is created before the f() function, and thus can be read from inside the f() function.

```
# Create the x variable and set to 44
x = 44

# Define a simple function that prints x
def f():
    print(x)

# Call the function
f()
```

Variables created ahead of a function may be read inside of the function only if the function does not change the value. This code, very similar to the code above, will fail. The computer will claim it doesn’t know what x is.

```
# Create the x variable and set to 44
x = 44
```

(continues on next page)

(continued from previous page)

```
# Define a simple function that prints x
def f():
    x += 1
    print(x)

# Call the function
f()
```

Other languages have more complex rules around the creation of variables and scope than Python does. Because Python is straight-forward it is a good introductory language.

1.8.6 Pass-by-Copy

Functions pass their values by creating a copy of the original. For example:

```
# Define a simple function that prints x
def f(x):
    x += 1
    print(x)

# Set y
y = 10
# Call the function
f(y)
# Print y to see if it changed
print(y)
```

The value of y does not change, even though the f() function increases the value passed to it. Each of the variables listed as a parameter in a function is a brand new variable. The value of that variable is copied from where it is called.

This is reasonably straight forward in the prior example. Where it gets confusing is if both the code that calls the function and the function itself have variables named the same. The code below is identical to the prior listing, but rather than use y it uses x.

```
# Define a simple function that prints x
def f(x):
    x += 1
    print(x)

# Set x
x = 10
# Call the function
f(x)
# Print x to see if it changed
print(x)
```

The output is the same as the program that uses y. Even though both the function and the surrounding code use x for a variable name, there are actually two different variables. There is the variable x that exists inside of the function, and a different variable x that exists outside the function.

1.8.7 Functions Calling Functions

For each of the examples below, think about what would print. Check to see if you are right. If you didn't guess correctly, spend the time to understand why.

Example 1

In this example, note that if you don't use a function, it doesn't run.

```
# Example 1
def a():
    print("A")

def b():
    print("B")

def c():
    print("C")

a()
```

Example 2

```
# Example 2
def a():
    b()
    print("A")

def b():
    c()
    print("B")

def c():
    print("C")

a()
```

Example 3

```
# Example 3
def a():
    print("A")
    b()

def b():
    print("B")
```

(continues on next page)

(continued from previous page)

```
c()

def c():
    print("C")

a()
```

Example 4

```
# Example 4
def a():
    print("A start")
    b()
    print("A end")

def b():
    print("B start")
    c()
    print("B end")

def c():
    print("C start and end")

a()
```

Example 5

```
# Example 5
def a(x):
    print("A start, x =", x)
    b(x + 1)
    print("A end, x =", x)

def b(x):
    print("B start, x =", x)
    c(x + 1)
    print("B end, x =", x)

def c(x):
    print("C start and end, x =", x)

a(5)
```

Example 6

While line 3 of this example increases `x`, the `x` variable in the function is a different variable than the `x` that is in the rest of the program. So that `x` never changes.

```
# Example 6
def a(x):
    x = x + 1

x = 3
a(x)

print(x)
```

Example 7

This example is similar to the prior example, but we `return x` at the end. Turns out, it doesn't matter. Because we never do anything with the return value. So the global variable `x` still doesn't increase. See the next example.

```
# Example 7
def a(x):
    x = x + 1
    return x

x = 3
a(x)

print(x)
```

Example 8

This example take the value returned from `a` and stores it back into `x`. How? By doing `x = a(x)` instead of just `a(x)`.

```
# Example 8
def a(x):
    x = x + 1
    return x

x = 3
x = a(x)

print(x)
```

Example 9

```
# Example 9
def a(x, y):
    x = x + 1
    y = y + 1
```

(continues on next page)

(continued from previous page)

```
print(x, y)

x = 10
y = 20
a(y, x)
```

Example 10

While you can have two `return` statements in a function, once you hit the first `return` the function ends. In this case, `return y` never runs, because we already returned from the function in the prior line.

```
# Example 10
def a(x, y):
    x = x + 1
    y = y + 1
    return x
    return y

x = 10
y = 20
z = a(x, y)

print(z)
```

Example 11

This is not something you can do in every programming language. You can return two values by using a comma and listing them.

```
# Example 11
def a(x, y):
    x = x + 1
    y = y + 1
    return x, y

x = 10
y = 20
z = a(x, y)

print(z)
```

Example 12

If you return two values out of a function, you can capture them this way.

```
# Example 12
def a(x, y):
    x = x + 1
    y = y + 1
```

(continues on next page)

(continued from previous page)

```

return x, y

x = 10
y = 20
x2, y2 = a(x, y) # Most computer languages don't support this

print(x2)
print(y2)

```

Example 13

```

# Example 13
def a(my_data):
    print("function a, my_data = ", my_data)
    my_data = 20
    print("function a, my_data = ", my_data)

my_data = 10

print("global scope, my_data =", my_data)
a(my_data)
print("global scope, my_data =", my_data)

```

Example 14

We will talk more about these next two examples when we talk about “lists” and “classes” later. These examples don’t operate like you might expect at first. Take a look and see what is different. We’ll explain why it works differently later.

```

# Example 14
def a(my_list):
    print("function a, list = ", my_list)
    my_list = [10, 20, 30]
    print("function a, list = ", my_list)

my_list = [5, 2, 4]

print("global scope, list =", my_list)
a(my_list)
print("global scope, list =", my_list)

```

Example 15

```

# Example 15
# New concept!
# Covered in more detail in a later chapter
def a(my_list):
    print("function a, list = ", my_list)

```

(continues on next page)

(continued from previous page)

```

my_list[0] = 1000
print("function a, list = ", my_list)

my_list = [5, 2, 4]

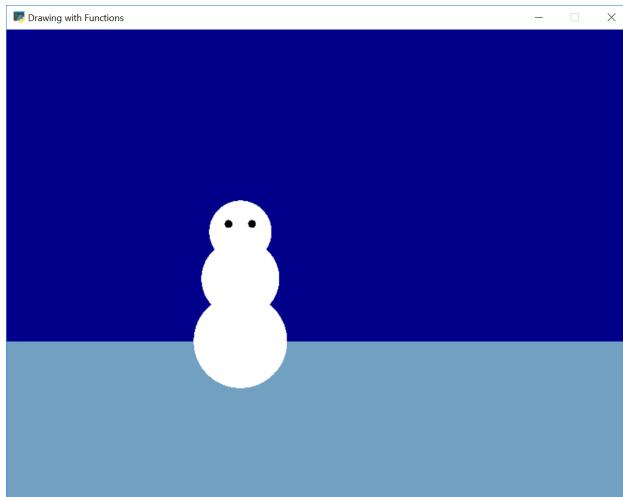
print("global scope, list =", my_list)
a(my_list)
print("global scope, list =", my_list)

```

1.9 Drawing With Functions

Here is a set of examples where we take a program that already exists and put everything in functions.

First the original program:



```

1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6
7 arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing with Functions")
8 arcade.set_background_color(arcade.color.DARK_BLUE)
9 arcade.start_render()
10
11 # Draw the ground
12 arcade.draw_lrtb_rectangle_filled(0, SCREEN_WIDTH, SCREEN_HEIGHT / 3, 0, arcade.color.
13     ↪AIR_SUPERIORITY_BLUE)
14
15 # Draw a snow person
16
17 # Snow
18 arcade.draw_circle_filled(300, 200, 60, arcade.color.WHITE)
19 arcade.draw_circle_filled(300, 280, 50, arcade.color.WHITE)
20 arcade.draw_circle_filled(300, 340, 40, arcade.color.WHITE)

```

(continues on next page)

(continued from previous page)

```

21 # Eyes
22 arcade.draw_circle_filled(285, 350, 5, arcade.color.BLACK)
23 arcade.draw_circle_filled(315, 350, 5, arcade.color.BLACK)
24
25 # Finish and run
26 arcade.finish_render()
27 arcade.run()

```

1.9.1 Make The main Function

Next, create a `main()` function. Put everything in it, and call the main function.

```

1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6
7 def main():
8     arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing with Functions")
9     arcade.set_background_color(arcade.color.DARK_BLUE)
10    arcade.start_render()
11
12    # Draw the ground
13    arcade.draw_lrtb_rectangle_filled(0, SCREEN_WIDTH, SCREEN_HEIGHT / 3, 0, arcade.
14    ↪color.AIR_SUPERIORITY_BLUE)
15
16    # Draw a snow person
17
18    # Snow
19    arcade.draw_circle_filled(300, 200, 60, arcade.color.WHITE)
20    arcade.draw_circle_filled(300, 280, 50, arcade.color.WHITE)
21    arcade.draw_circle_filled(300, 340, 40, arcade.color.WHITE)
22
23    # Eyes
24    arcade.draw_circle_filled(285, 350, 5, arcade.color.BLACK)
25    arcade.draw_circle_filled(315, 350, 5, arcade.color.BLACK)
26
27    # Finish and run
28    arcade.finish_render()
29    arcade.run()
30
31 # Call the main function to get the program started.
32 main()

```

When you do this, run your program and make sure it still works before proceeding.

1.9.2 Make The Drawing Functions

Next, pick an item to move to a function. Start with an easy one if you have it. I chose grass because it was only one line of code, and I wasn't going to ever try to position it with x, y.

```
1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6
7 def draw_grass():
8     """ Draw the ground """
9     arcade.draw_lrtb_rectangle_filled(0, SCREEN_WIDTH, SCREEN_HEIGHT / 3, 0, arcade.
10                                     color.AIR_SUPERIORITY_BLUE)
11
12 def main():
13     arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing with Functions")
14     arcade.set_background_color(arcade.color.DARK_BLUE)
15     arcade.start_render()
16
17     draw_grass()
18
19     # Draw a snow person
20
21     # Snow
22     arcade.draw_circle_filled(300, 200, 60, arcade.color.WHITE)
23     arcade.draw_circle_filled(300, 280, 50, arcade.color.WHITE)
24     arcade.draw_circle_filled(300, 340, 40, arcade.color.WHITE)
25
26     # Eyes
27     arcade.draw_circle_filled(285, 350, 5, arcade.color.BLACK)
28     arcade.draw_circle_filled(315, 350, 5, arcade.color.BLACK)
29
30     # Finish and run
31     arcade.finish_render()
32     arcade.run()
33
34
35 # Call the main function to get the program started.
36 main()
```

Then, I took a more complex shape and put it in a function.

```
1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6
7 def draw_grass():
8     """ Draw the ground """
9     arcade.draw_lrtb_rectangle_filled(0, SCREEN_WIDTH, SCREEN_HEIGHT / 3, 0, arcade.
10                                     color.AIR_SUPERIORITY_BLUE)
11
12 def draw_snow_person():
13     """ Draw a snow person """
14
15     # Snow
16     arcade.draw_circle_filled(300, 200, 60, arcade.color.WHITE)
```

(continues on next page)

(continued from previous page)

```

17     arcade.draw_circle_filled(300, 280, 50, arcade.color.WHITE)
18     arcade.draw_circle_filled(300, 340, 40, arcade.color.WHITE)

19
20     # Eyes
21     arcade.draw_circle_filled(285, 350, 5, arcade.color.BLACK)
22     arcade.draw_circle_filled(315, 350, 5, arcade.color.BLACK)

23
24
25 def main():
26     arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing with Functions")
27     arcade.set_background_color(arcade.color.DARK_BLUE)
28     arcade.start_render()

29
30     draw_grass()
31     draw_snow_person()

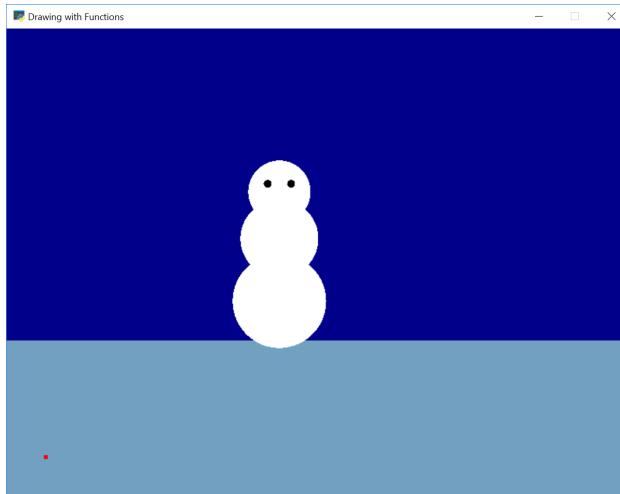
32
33     # Finish and run
34     arcade.finish_render()
35     arcade.run()

36
37
38 # Call the main function to get the program started.
39 main()

```

But this draws the snowman only at one spot. I want to draw lots of snowmen, anywhere I put them!

To do this, let's add an x and y:



```

1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6
7 def draw_grass():
8     """ Draw the ground """
9     arcade.draw_lrtb_rectangle_filled(0, SCREEN_WIDTH, SCREEN_HEIGHT / 3, 0, arcade.
10                                     color.AIR_SUPERIORITY_BLUE)

```

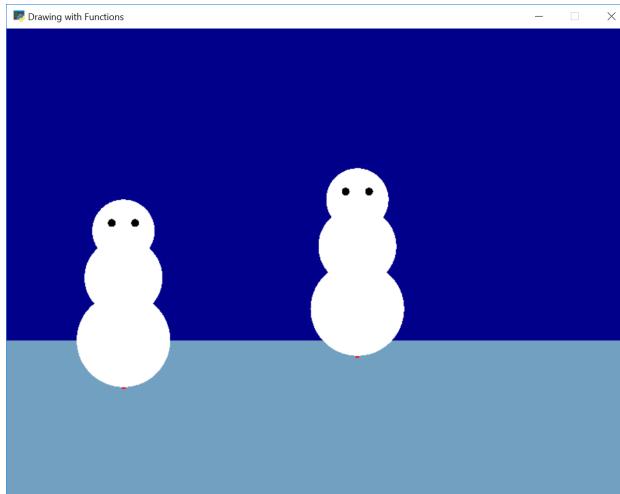
(continues on next page)

(continued from previous page)

```
11
12 def draw_snow_person(x, y):
13     """ Draw a snow person """
14
15     # Draw a point at x, y for reference
16     arcade.draw_point(x, y, arcade.color.RED, 5)
17
18     # Snow
19     arcade.draw_circle_filled(300 + x, 200 + y, 60, arcade.color.WHITE)
20     arcade.draw_circle_filled(300 + x, 280 + y, 50, arcade.color.WHITE)
21     arcade.draw_circle_filled(300 + x, 340 + y, 40, arcade.color.WHITE)
22
23     # Eyes
24     arcade.draw_circle_filled(285 + x, 350 + y, 5, arcade.color.BLACK)
25     arcade.draw_circle_filled(315 + x, 350 + y, 5, arcade.color.BLACK)
26
27
28 def main():
29     arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing with Functions")
30     arcade.set_background_color(arcade.color.DARK_BLUE)
31     arcade.start_render()
32
33     draw_grass()
34     draw_snow_person(50, 50)
35
36     # Finish and run
37     arcade.finish_render()
38     arcade.run()
39
40
41 # Call the main function to get the program started.
42 main()
```

But that's not perfect. If you'll note, I added a dot at the x and y. The snowman draws way off from the dot, because originally I didn't try to draw it at 0, 0. I need to recenter the snowman on the dot.

We need to re-center the shape onto the spot we are drawing. Typically you'll need to subtract from all the x and y values the same amount.



```

1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6
7 def draw_grass():
8     """ Draw the ground """
9     arcade.draw_lrtb_rectangle_filled(0, SCREEN_WIDTH, SCREEN_HEIGHT / 3, 0, arcade.
10                                     color.AIR_SUPERIORITY_BLUE)
11
12 def draw_snow_person(x, y):
13     """ Draw a snow person """
14
15     # Draw a point at x, y for reference
16     arcade.draw_point(x, y, arcade.color.RED, 5)
17
18     # Snow
19     arcade.draw_circle_filled(x, 60 + y, 60, arcade.color.WHITE)
20     arcade.draw_circle_filled(x, 140 + y, 50, arcade.color.WHITE)
21     arcade.draw_circle_filled(x, 200 + y, 40, arcade.color.WHITE)
22
23     # Eyes
24     arcade.draw_circle_filled(x - 15, 210 + y, 5, arcade.color.BLACK)
25     arcade.draw_circle_filled(x + 15, 210 + y, 5, arcade.color.BLACK)
26
27
28 def main():
29     arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing with Functions")
30     arcade.set_background_color(arcade.color.DARK_BLUE)
31     arcade.start_render()
32
33     draw_grass()
34     draw_snow_person(150, 140)
35     draw_snow_person(450, 180)
36
37     # Finish and run
38     arcade.finish_render()
39     arcade.run()
40
41
42 # Call the main function to get the program started.
43 main()

```

1.9.3 How To Animate A Drawing Function

We can animate our drawing if we want. Here are the steps.

Create An `on_draw` Method

Right now our program only draws our image once. We need to move all the drawing code in our `main` to an `on_draw` function. Then we'll tell the computer to draw that over and over.

Continuing from our last example, our program will look like:

```
1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6
7 def draw_grass():
8     """ Draw the ground """
9     arcade.draw_lrtb_rectangle_filled(0, SCREEN_WIDTH, SCREEN_HEIGHT / 3, 0, arcade.
10                                     color.AIR_SUPERIORITY_BLUE)
11
12 def draw_snow_person(x, y):
13     """ Draw a snow person """
14
15     # Draw a point at x, y for reference
16     arcade.draw_point(x, y, arcade.color.RED, 5)
17
18     # Snow
19     arcade.draw_circle_filled(x, 60 + y, 60, arcade.color.WHITE)
20     arcade.draw_circle_filled(x, 140 + y, 50, arcade.color.WHITE)
21     arcade.draw_circle_filled(x, 200 + y, 40, arcade.color.WHITE)
22
23     # Eyes
24     arcade.draw_circle_filled(x - 15, 210 + y, 5, arcade.color.BLACK)
25     arcade.draw_circle_filled(x + 15, 210 + y, 5, arcade.color.BLACK)
26
27
28 def on_draw(delta_time):
29     """ Draw everything """
30     arcade.start_render()
31
32     draw_grass()
33     draw_snow_person(150, 140)
34     draw_snow_person(450, 180)
35
36
37 def main():
38     arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing with Functions")
39     arcade.set_background_color(arcade.color.DARK_BLUE)
40
41     # Call on_draw every 60th of a second.
42     arcade.schedule(on_draw, 1/60)
43     arcade.run()
44
45
46 # Call the main function to get the program started.
47 main()
```

Do this with your own program. Nothing will move, but it should still run.

Add Variable To Control Where We Draw Our Item

Next, we are going to create a variable inside of the `on_draw` function. This variable will hold our `x` value. Each time we call `on_draw`, we'll change `x` so that it moves to the right.

```

1 import arcade
2
3 SCREEN_WIDTH = 800
4 SCREEN_HEIGHT = 600
5
6
7 def draw_grass():
8     """ Draw the ground """
9     arcade.draw_lrtb_rectangle_filled(0, SCREEN_WIDTH, SCREEN_HEIGHT / 3, 0, arcade.
10                                     color.AIR_SUPERIORITY_BLUE)
11
12 def draw_snow_person(x, y):
13     """ Draw a snow person """
14
15     # Draw a point at x, y for reference
16     arcade.draw_point(x, y, arcade.color.RED, 5)
17
18     # Snow
19     arcade.draw_circle_filled(x, 60 + y, 60, arcade.color.WHITE)
20     arcade.draw_circle_filled(x, 140 + y, 50, arcade.color.WHITE)
21     arcade.draw_circle_filled(x, 200 + y, 40, arcade.color.WHITE)
22
23     # Eyes
24     arcade.draw_circle_filled(x - 15, 210 + y, 5, arcade.color.BLACK)
25     arcade.draw_circle_filled(x + 15, 210 + y, 5, arcade.color.BLACK)
26
27
28 def on_draw(delta_time):
29     """ Draw everything """
30     arcade.start_render()
31
32     draw_grass()
33     draw_snow_person(on_draw.snow_person1_x, 140)
34     draw_snow_person(450, 180)
35
36     # Add one to the x value, making the snow person move right
37     # Negative numbers move left. Larger numbers move faster.
38     on_draw.snow_person1_x += 1
39
40
41 # Create a value that our on_draw.snow_person1_x will start at.
42 on_draw.snow_person1_x = 150
43
44
45 def main():
46     arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing with Functions")
47     arcade.set_background_color(arcade.color.DARK_BLUE)
48
49     # Call on_draw every 60th of a second.
50     arcade.schedule(on_draw, 1/60)
51     arcade.run()
52
53
54 # Call the main function to get the program started.
55 main()

```

For more information, see the [Bouncing Rectangle Example](#).

1.10 If Statements

How do we tell if a player has beat the high score? How can we tell if he has run out of lives? How can we tell if she has the key required to open the locked door?

What we need is the `if` statement. The `if` statement is also known as a *conditional statement*. (You can use the term “conditional statement” when you want to impress everyone how smart you are.) The `if` statement allows a computer to make a decision. Is it hot outside? Has the spaceship reached the edge of the screen? Has too much money been withdrawn from the account? A program can test for these conditions with the `if` statement.

1.10.1 Basic Comparisons

Here are a few examples of `if` statements. The first section sets up two variables (`a` and `b`) for use in the `if` statements. Then two `if` statements show how to compare the variables to see if one is greater than the other.

Listing 1: Example `if` statements: less than, greater than

```
1 # Variables used in the example ``if`` statements
2 a = 4
3 b = 5
4
5 # Basic comparisons
6 if a < b:
7     print("a is less than b")
8
9 if a > b:
10    print("a is greater than b")
11
12 print("Done")
```

Listing 2: Output

```
a is less than b
Done
```

Since `a` is less than `b`, the first statement will print out if this code is run. If the variables `a` and `b` were both equal to 4, then neither of the two `if` statements above would print anything out. The number 4 is not greater than 4, so the `if` statement would fail.

To show the flow of a program a *flowchart* may be used. Most people can follow a flowchart even without an introduction to programming. See how well you can understand the figure below.

This book skips an in-depth look at flowcharting because it is boring. But if you want to be a superstar programmer, please read more about it at:

<http://en.wikipedia.org/wiki/Flowchart>

The prior example checked for greater than or less than. Numbers that were equal would not pass the test. To check for a values greater than or equal, the following examples show how to do this:

Listing 3: Example `if` statements: less than or equal, greater than or equal

```
1 if a <= b:
2     print("a is less than or equal to b")
3
```

(continues on next page)

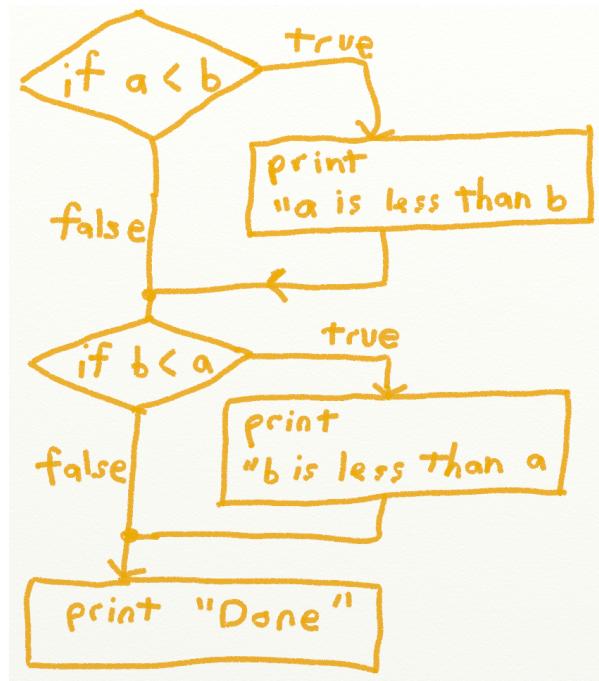


Fig. 7: Flowchart

(continued from previous page)

```

4 if a >= b:
5   print("a is greater than or equal to b")
  
```

The `<=` and `>=` symbols must be used in order, and there may not be a space between them. For example, `=<` will not work, nor will `< =`.

When writing these statements out on a test, some students like to use the `symbol`. For example:

```
if a b:
```

This `symbol` doesn't actually work in a program. Plus most people don't know how to easily type it on the keyboard. (Just in case you are curious, to type it hold down the 'alt' key while typing 243 on the number pad.) So when writing out code, remember that it is `<=` and not `. Many people lose points on tests for this reason; don't be that person.`

The next set of code checks to see if two items are equal or not. The operator for equal is `==` and the operator for not equal is `!=`. Here they are in action.

Listing 4: Example if statements: equal not equal

```
1 # Equal
2 if a == b:
3     print("a is equal to b")
4
5 # Not equal
6 if a != b:
7     print("a and b are not equal")
```

Attention: Learn when to use = and ==.

It is very easy to mix up when to use == and =. Use == if you are asking if they are equal, use = if you are assigning a value.

The two most common mistakes in mixing the = and == operators are demonstrated below:

```
1 # This is wrong
2 a == 1
3
4 # This is also wrong
5 if a = 1:
6     print("A is one")
```

Stop! Please take a moment to go back and carefully study the last two code examples. Save time later by making sure you understand when to use = and ==. Don't guess.

1.10.2 Indentation

Indentation matters. Each line under the if statement that is indented will only be executed if the statement is True:

```
1 if a == 1:
2     print("If a is one, this will print.")
3     print("So will this.")
4     print("And this.")
5
6 print("This will always print because it is not indented.")
```

Indentation must be the same. This code doesn't work.

```
1 if a == 1:
2     print("Indented two spaces.")
3     print("Indented four. This will generate an error.")
4     print("The computer will want you to make up your mind.")
```

Once an if statement has been finished, it is not possible to re-indent to go back to it. The test has to be performed again.

```
1 if a == 1:
2     print("If a is one, this will print.")
3     print("So will this.")
4
5 print("This will always print because it is not indented.")
6     print("This will generate an error. Why it is indented?")
```

1.10.3 Using And/Or

An `if` statement can check multiple conditions by chaining together comparisons with `and` and `or`. These are also considered to be *operators* just like `+` or `-` are.

Listing 5: Example `if` statements, using “and” and “or”

```

1 # And
2 if a < b and a < c:
3     print("a is less than b and c")
4
5 # Non-exclusive or
6 if a < b or a < c:
7     print("a is less than either b or c (or both)")
```

Hint: Repeat yourself please.

A common mistake is to omit a variable when checking it against multiple conditions. The code below does not work because the computer does not know what to check against the variable `c`. It will not assume to check it against `a`.

```

1 # This is not correct
2 if a < b or < c:
3     print("a is less than b and c")
```

1.10.4 Boolean Variables

Python supports Boolean variables. What are Boolean variables? Boolean variables can store either a `True` or a value of `False`. Boolean algebra was developed by George Boole back in 1854. If only he knew how important his work would become as the basis for modern computer logic!

An `if` statement needs an expression to evaluate to `True` or `False`. What may seem odd is that it does not actually need to do any comparisons if a variable already evaluates to `True` or `False`.

Listing 6: If statements and Boolean data types

```

1 # Boolean data type. This is legal!
2 a = True
3 if a:
4     print("a is true")
```

Back when I was in school it was popular to say some false statement. Wait three seconds, then shout “NOT!” Well, even your computer thinks that is lame. If you are going to do that, you have to start with the `not` operator. The following code uses the `not` to flip the value of `a` between `true` and `false`.

Because `not` is an operator and not a function, parentheses aren’t necessary.

Listing 7: The `not` operator example 2

```

1 # How to use the not function
2 if not a:
3     print("a is false")
```

It is also possible to use Boolean variables with `and` and `or` operators.

Listing 8: Using “and” with Boolean variables

```
1 a = True
2 b = False
3
4 if a and b:
5     print("a and b are both true")
```

Attention: Who knew True/False could be hard?

It is also possible to assign a variable to the result of a comparison. In the code below, the variables `a` and `b` are compared. If they are equal, `c` will be `True`, otherwise `c` will be `False`.

Listing 9: Assigning values to Boolean data types

```
1 a = 3
2 b = 3
3
4 # This next line is strange-looking, but legal.
5 # c will be true or false, depending if
6 # a and b are equal.
7 c = a == b
8
9 # Prints value of c, in this case True
10 print(c)
```

Hint: Zero means False. Everything else is True.

It is possible to create an `if` statement with a condition that does not evaluate to `True` or `False`. This is not usually desired, but it is important to understand how the computer handles these values when searching for problems. The statement below is legal and will cause the text to be printed out because the values in the `if` statement are non-zero:

```
1 if 1:
2     print("1")
3 if "A":
4     print("A")
```

The code below will not print out anything because the value in the `if` statement is zero which is treated as `False`. Any value other than zero is considered `True`.

```
1 if 0:
2     print("Zero")
```

1.10.5 The `input` Function

Rather than hard-coding values into our program, we can use the `input` function to ask the user to type something in. The `input` function is reasonably simple to use:

```
1 temperature = input("What is the temperature in Fahrenheit? ")
2 print("You said the temperature was " + temperature + ".")
```

As a parameter to `input`, just give it text to use as a prompt. Whatever the user types in, is stored in the variable on the left.

Note that there is a question mark and a space at the end of that string. If you didn't have this, what the user types in will run right up next to the prompt. That looks terrible. The user is tempted to type a space as the first part of their input, which complicates things for us later.

There's one more thing we have to learn. We can't take what the user types in and compare it to a number. This program:

```
1 temperature = input("What is the temperature in Fahrenheit? ")
2 if temperature > 90:
3     print("It is hot outside.")
```

...will fail with the error:

```
TypeError: '>' not supported between instances of 'str' and 'int'
```

Whatever the user types in is stored as text. We also call text a "string" because to the computer it is just a string of characters. This is different than a number, and the computer does not know how to compare them.

Therefore, we need to convert the input into a number. We can do with either the `int` for integers, or the `float` function for numbers with a decimal.

For example:

```
1 # Get input from the user
2 temperature = input("What is the temperature in Fahrenheit? ")
3
4 # Convert the input to an integer
5 temperature = int(temperature)
6
7 # Do our comparison
8 if temperature > 90:
9     print("It is hot outside.")
```

You can also do it in one step by nesting the functions:

```
1 # Get input from the user
2 temperature = int(input("What is the temperature in Fahrenheit? "))
3
4 # Do our comparison
5 if temperature > 90:
6     print("It is hot outside.")
```

1.10.6 Else and Else If

Below is code that will get the temperature from the user and print if it is hot.

```
1 temperature = int(input("What is the temperature in Fahrenheit? "))
2 if temperature > 90:
3     print("It is hot outside")
4     print("Done")
```

If the programmer wants code to be executed if it is not hot, she can use the `else` statement. Notice how the `else` is lined up with the `i` in the `if` statement, and how it is followed by a colon just like the `if` statement.

In the case of an if...else statement, one block of code will always be executed. The first block will be executed if the statement evaluates to True, the second block if it evaluates to False.

Listing 10: Example if/else statement

```
1 temperature = int(input("What is the temperature in Fahrenheit? "))
2 if temperature > 90:
3     print("It is hot outside")
4 else:
5     print("It is not hot outside")
6 print("Done")
```

It is possible to chain several if statements together using the else...if statement. Python abbreviates this as elif.

Listing 11: Example if/elif/else statement

```
1 temperature = int(input("What is the temperature in Fahrenheit? "))
2 if temperature > 90:
3     print("It is hot outside")
4 elif temperature < 30:
5     print("It is cold outside")
6 else:
7     print("It is not hot outside")
8 print("Done")
```

In the code below, the program will output “It is hot outside” even if the user types in 120 degrees. Why? How can the code be fixed?

If you can't figure it out, see the video.

Listing 12: Example of improper ordering if/elif/else

```
1 temperature = int(input("What is the temperature in Fahrenheit? "))
2 if temperature > 90:
3     print("It is hot outside")
4 elif temperature > 110:
5     print("Oh man, you could fry eggs on the pavement!")
6 elif temperature < 30:
7     print("It is cold outside")
8 else:
9     print("It is ok outside")
10 print("Done")
```

1.10.7 Text Comparisons

It is possible to use an if statement to check text.

Listing 13: Case sensitive text comparison

```
1 user_name = input("What is your name? ")
2 if user_name == "Paul":
3     print("You have a nice name.")
4 else:
5     print("Your name is ok.")
```

The prior example will only match if the user enters “Paul”. It will not work if the user enters “paul” or “PAUL”.

A common mistake is to forget the quotes around the string being compared. In the example below, the computer will think that Paul is a variable that stores a value. It will flag an error because it has no idea what is stored in the variable Paul.

Listing 14: Incorrect comparison

```

1 user_name = input("What is your name? ")
2 if user_name == Paul: # This does not work because quotes are missing
3     print("You have a nice name.")
4 else:
5     print("Your name is ok.")

```

Multiple Text Possibilities

When comparing a variable to multiple possible strings of text, it is important to remember that the comparison must include the variable. For example:

```

1 # This does not work! It will always be true
2 if user_name == "Paul" or "Mary":
3 Instead, the code should read:
4
5 # This does work
6 if user_name == "Paul" or user_name == "Mary":

```

This is because any value other than zero, the computer assumes to mean True. So to the computer “Mary” is the same thing as True and so it will run the code in the if statement.

Case Insensitive Comparisons

If the program needs to match regardless as to the case of the text entered, the easiest way to do that is to convert everything to lower case. This can be done with the lower command.

Attention: Learn to be insensitive.

The example below will take whatever the user enters, convert it to lower case, and then do the comparison. Important: Don’t compare it against a string that has uppercase. If the user input is converted to lowercase, then compared against uppercase letters, there is no way a match can occur.

Listing 15: Case-insensitive text comparison

```

1 user_name = input("What is your name? ")
2 if user_name.lower() == "paul":
3     print("You have a nice name.")
4 else:
5     print("Your name is ok.")

```

1.10.8 Example if Statements

The next set of example code below runs through all the concepts talked about earlier. The on-line video traces through each line of code and explains how it works

```
1 # Sample Python/Pygame Programs
2 # Simpson College Computer Science
3 # http://programarcadegames.com/
4 # http://simpson.edu/computer-science/
5
6 # Explanation video: http://youtu.be/pDpNSck2aXQ
7
8 # Variables used in the example if statements
9 a = 4
10 b = 5
11 c = 6
12
13 # Basic comparisons
14 if a < b:
15     print("a is less than b")
16
17 if a > b:
18     print("a is greater than b")
19
20 if a <= b:
21     print("a is less than or equal to b")
22
23 if a >= b:
24     print("a is greater than or equal to b")
25
26 # NOTE: It is very easy to mix when to use == and =
27 # Use == if you are asking if they are equal, use =
28 # if you are assigning a value.
29 if a == b:
30     print("a is equal to b")
31
32 # Not equal
33 if a != b:
34     print("a and b are not equal")
35
36 # And
37 if a < b and a < c:
38     print("a is less than b and c")
39
40 # Non-exclusive or
41 if a < b or a < c:
42     print("a is less than either a or b (or both)")
43
44
45 # Boolean data type. This is legal!
46 a = True
47 if a:
48     print("a is true")
49
50 if not a:
51     print("a is false")
52
53 a = True
54 b = False
55
56 if a and b:
57     print("a and b are both true")
```

(continues on next page)

(continued from previous page)

```

58
59 a = 3
60 b = 3
61 c = a == b
62 print(c)
63
64 # These are also legal and will trigger as being true because
65 # the values are not zero:
66 if 1:
67     print("1")
68 if "A":
69     print("A")
70
71 # This will not trigger as true because it is zero.
72 if 0:
73     print("Zero")
74
75 # Comparing variables to multiple values.
76 # The first if statement appears to work, but it will always
77 # trigger as true even if the variable a is not equal to b.
78 # This is because "b" by itself is considered true.
79 a = "c"
80 if a == "B" or "b":
81     print("a is equal to b. Maybe.")
82
83 # This is the proper way to do the if statement.
84 if a == "B" or a == "b":
85     print("a is equal to b.")
86
87 # Example 1: If statement
88 temperature = int(input("What is the temperature in Fahrenheit? "))
89 if temperature > 90:
90     print("It is hot outside")
91 print("Done")
92
93 # Example 2: Else statement
94 temperature = int(input("What is the temperature in Fahrenheit? "))
95 if temperature > 90:
96     print("It is hot outside")
97 else:
98     print("It is not hot outside")
99 print("Done")
100
101 # Example 3: Else if statement
102 temperature = int(input("What is the temperature in Fahrenheit? "))
103 if temperature > 90:
104     print("It is hot outside")
105 elif temperature < 30:
106     print("It is cold outside")
107 else:
108     print("It is not hot outside")
109 print("Done")
110
111 # Example 4: Ordering of statements
112 # Something with this is wrong. What?
113 temperature = int(input("What is the temperature in Fahrenheit? "))
114 if temperature > 90:

```

(continues on next page)

(continued from previous page)

```

115     print("It is hot outside")
116 elif temperature > 110:
117     print("Oh man, you could fry eggs on the pavement!")
118 elif temperature < 30:
119     print("It is cold outside")
120 else:
121     print("It is ok outside")
122 print("Done")

123
124 # Comparisons using string/text
125 # The input statement will ask the user for input and put it in user_name.
126 user_name = input("What is your name? ")
127 if user_name == "Paul":
128     print("You have a nice name.")
129 else:
130     print("Your name is ok.")

```

1.11 Guessing Games with Random Numbers and Loops

Our next step is how to loop a section of code. Most games “loop.” They repeat the same code over and over. For example the number guessing game below loops for each guess that the user makes:

```

Hi! I'm thinking of a random number between 1 and 100.
--- Attempt 1
Guess what number I am thinking of: 50
Too high.
--- Attempt 2
Guess what number I am thinking of: 25
Too high.
--- Attempt 3
Guess what number I am thinking of: 17
Too high.
--- Attempt 4
Guess what number I am thinking of: 9
Too low.
--- Attempt 5
Guess what number I am thinking of: 14
Too high.
--- Attempt 6
Guess what number I am thinking of: 12
Too high.
--- Attempt 7
Guess what number I am thinking of: 10
Too low.
Aw, you ran out of tries. The number was 11.

```

(Code for this program is below: [Number Guessing Game Example](#))

Wait, what does this looping have to do with graphics and video games? A lot. Each *frame* the game displays is one time through a loop. You may be familiar with the frames-per-second (FPS) statistic that games show. The FPS represents the number of times the computer updates the screen each second. The higher the rate, the smoother the game. (Although an FPS rate past 60 is faster than most screens can update, so there isn’t much point to push it past that.) The figure below shows the game Eve Online and a graph showing how many frames per second the computer is able to display.

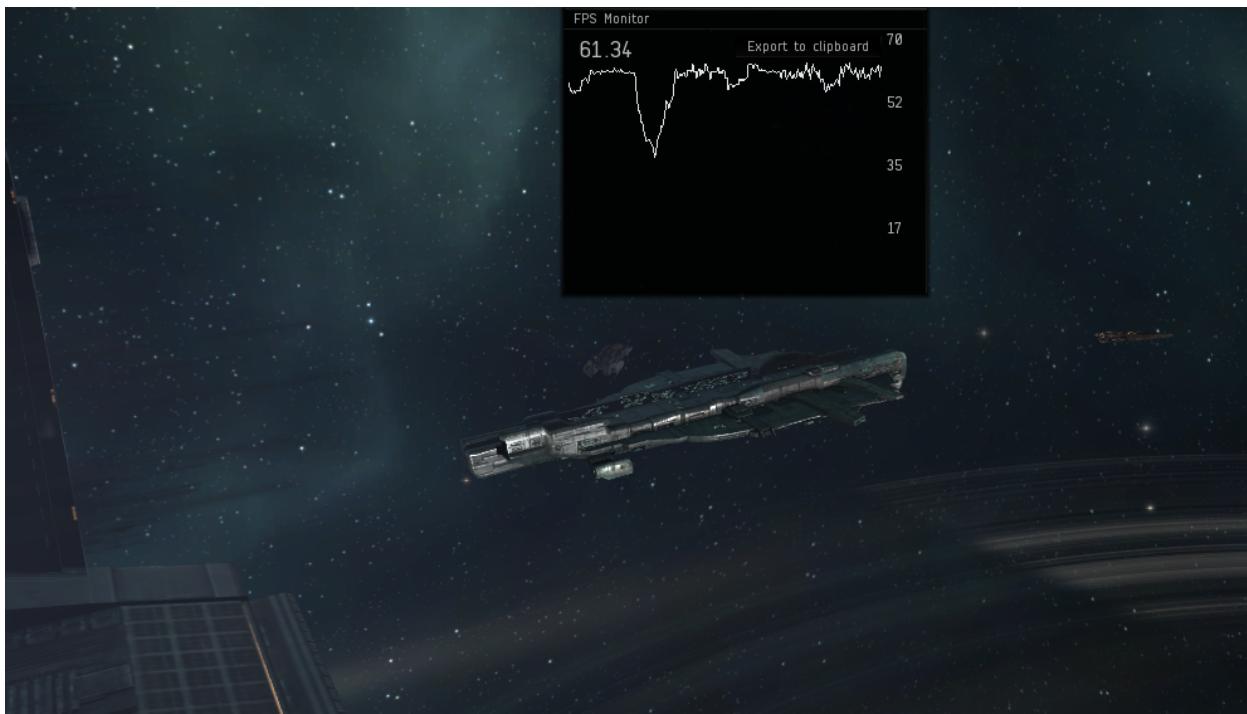


Fig. 8: FPS in video games

The loop in these games works like the flowchart in the figure below. Despite the complexities of modern games, the inside of this loop is similar to a calculator program. Get user input. Perform calculations. Output the result. In a video game, we try to repeat this up to 60 times per second.

Fig. 9: Game loop

There can even be loops inside of other loops. A real “loop the loop.” Take a look at the “Draw Everything” box in Figure 4.2. This set of code loops through and draws each object in the game. That loop is inside of the larger loop that draws each frame of the game, which looks like the figure below.

Fig. 10: Draw everything loop

There are two major types of loops in Python, `for` loops and `while` loops. If you want to repeat a certain number of times, use a `for` loop. If you want to repeat until something happens (like the user hits the quit button) then use a `while` loop.

For example, a `for` loop can be used to print all student records since the computer knows how many students there are. A `while` loop would need to be used to check for when a user hits the mouse button since the computer has no idea how long it will have to wait.

1.11.1 For Loops

The `for` loop example below runs the `print` statement five times. It could just as easily run 100 or 1,000,000 times just by changing the 5 to the desired number of times to loop. Note the similarities of how the `for` loop is written to the if statement. Both end in a colon, and both use indentation to specify which lines are affected by the statement.

Listing 16: Loop to print five times

```
1 for i in range(5):
2     print("I will not chew gum in class.")
```

Output:

```
I will not chew gum in class.
```

The `i` on line 1 is a variable that keeps track of how many times the program has looped. It is a new variable and can be named any legal variable name. Programmers often use `i` as for the variable name, because the `i` is short for *increment*. This variable helps track when the loop should end.

The `range` function controls how many times the code in the loop is run. In this case, five times.

The next example code will print “Please,” five times and “Can I go to the mall?” only once. “Can I go to the mall?” is not indented so it is not part of the for loop and will not print until the for loop completes.

```
1 for i in range(5):
2     print("Please,")
3     print("Can I go to the mall?")
```

Output:

```
Please,
Please,
Please,
Please,
Please,
Can I go to the mall?
```

This next code example takes the prior example and indents line 3. This change will cause the program to print “Please,” and “Can I go to the mall?” five times. Since the statement has been indented “Can I go to the mall?” is now part of the for loop and will repeat five times just like the word “Please.”

```
1 for i in range(5):
2     print("Please,")
3     print("Can I go to the mall?")
```

Output:

```
Please,
Can I go to the mall?
```

You aren’t stuck using a specific number with the `range` function. This example asks the user how many times to print using the `input` function we talked about back in *The input Function*.

Listing 17: Loop according to the user input

```

1 # Ask the user how many times to print
2 repetitions = int(input("How many times should I repeat? "))
3
4 # Loop that many times
5 for i in range(repetitions):
6     print("I will not chew gum in class.")

```

Or you could write a function, and take in the value by a parameter:

Listing 18: Loop according to a function parameter

```

1 def print_about_gum(repetitions):
2
3     # Loop that many times
4     for i in range(repetitions):
5         print("I will not chew gum in class.")
6
7
8 def main():
9     print_about_gum(10)
10
11
12 main()

```

The code below will print the numbers 0 to 9. Notice that the loop starts at 0 and does not include the number 10. It is natural to assume that `range(10)` would include 10, but it stops just short of it.

Listing 19: Print the numbers 0 to 9

```

1 for i in range(10):
2     print(i)

```

Output:

```

0
1
2
3
4
5
6
7
8
9

```

A program does not need to name the variable `i`, it could be named something else. For example a programmer might use `line_number` if she was processing a text file.

If a programmer wants to go from 1 to 10 instead of 0 to 9, there are a couple ways to do it. The first way is to send the `range` function two numbers instead of one. The first number is the starting value, the second is just beyond the ending value.

It does take some practice to get used to the idea that the `for` loop will include the first number, but not the second number listed. The example below specifies a range of (1, 11), and the numbers 1 to 10 are printed. The starting number 1 is included, but not the ending number of 11.

Listing 20: Print the numbers 1 to 10, version 1

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

Output:

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

Another way to print the numbers 1 to 10 is to still use `range(10)` and have the variable `i` go from 0 to 9. But just before printing out the variable the programmer adds one to it. This also works to print the numbers 1 to 10. Either method works just fine.

Listing 21: Print the numbers 1 to 10, version 2

```
1 # Print the numbers 1 to 10.
2 for i in range(10):
3     print(i + 1)
```

Counting By Numbers Other Than One

If the program needs to count by 2's or use some other increment, that is easy. Just like before there are two ways to do it. The easiest is to supply a third number to the `range` function that tells it to count by 2's. The second way to do it is to go ahead and count by 1's, but multiply the variable by 2. The code example below shows both methods.

Listing 22: Two ways to print the even numbers 2 to 10

```
1 # Two ways to print the even numbers 2 to 10
2 for i in range(2,12,2):
3     print(i)
4
5 for i in range(5):
6     print((i + 1) * 2)
```

Output:

```
2
4
6
8
10
2
4
6
8
10
```

It is also possible to count backwards down towards zero by giving the `range` function a negative step. In the example below, start at 10, go down to but not including 0, and do it by -1 increments. The hardest part of creating these loops is to accidentally switch the start and end numbers. The program starts at the larger value, so it goes first. Normal for loops that count up start with the smallest value listed first in the `range` function.

Listing 23: Count down from 10 to 1

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

Output:

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

If the numbers that a program needs to iterate through don't form an easy pattern, it is possible to pull numbers out of a list. (A full discussion of lists is covered in a later chapter. This is just a preview of what you can do.)

Listing 24: Print numbers out of a list

```
1 for i in [2, 6, 4, 2, 4, 6, 7, 4]:
2     print(i)
```

This prints:

```
2
6
4
2
4
6
7
4
```

Nesting Loops

Try to predict what the code below will print. Then enter the code and see if you are correct.

```
1 # What does this print? Why?
2 for i in range(3):
3     print("a")
4 for j in range(3):
5     print("b")
```

This next block of code is almost identical to the one above. The second for loop has been indented one tab stop so that it is now nested inside of the first for loop. This changes how the code runs significantly. Try it and see.

```
1 # What does this print? Why?
2 for i in range(3):
3     print("a")
4     for j in range(3):
5         print("b")
6
7 print("Done")
```

I'm not going to tell you what the code does, go to a computer and see.

Keeping a Running Total

A common operation in working with loops is to keep a running total. This “running total” code pattern is used a lot in this book. Keep a running total of a score, total a person’s account transactions, use a total to find an average, etc. You might want to bookmark this code listing because we’ll refer back to it several times. In the code below, the user enters five numbers and the code totals up their values.

Listing 25: Keep a Running Total

```
1 total = 0
2 for i in range(5):
3     new_number = int(input("Enter a number: " ))
4     total = total + new_number
5 print("The total is: ", total)
```

Note that line 1 creates the variable `total`, and sets it to an initial amount of zero. It is easy to forget the need to create and initialize the variable to zero. Without it the computer will complain when it hits line 4. It doesn’t know how to add `n`“ew_number“ to `total` because `total` hasn’t been given a value yet.

A common mistake is to use `i` to `total` instead of `new_number`. Remember, we are keeping a running total of the values entered by the user, not a running total of the current loop count.

Speaking of the current loop count, we can use the loop count value to solve some mathematical operations. For example:

$$s = \sum_{n=1}^{100} n$$

If you aren’t familiar with this type of formula, it is just a fancy way of stating:

$$s = 1 + 2 + 3 + 4 + 5 \dots 98 + 99 + 100$$

The code below adds all the numbers from 1 to 100. It demonstrates a common pattern where a running total is kept inside of a loop. This also uses a separate variable `sum` to track the running total.

Listing 26: Sum all numbers 1 to 100

```
1 # What is the value of sum?
2 total = 0
3 for i in range(1, 101):
4     total = total + i
5 print(total)
```

Here’s a different variation. This takes five numbers from the user and counts the number of times the user enters a zero:

```

1 total = 0
2 for i in range(5):
3     new_number = int(input("Enter a number: "))
4     if new_number == 0:
5         total += 1
6 print("You entered a total of", total, "zeros")

```

A programmer that understands the nested for loops and running totals should be able to predict the output of the code below.

```

1 # What is the value of a?
2 a = 0
3 for i in range(10):
4     a = a + 1
5 print(a)
6
7 # What is the value of a?
8 a = 0
9 for i in range(10):
10    a = a + 1
11 for j in range(10):
12    a = a + 1
13 print(a)
14
15 # What is the value of a?
16 a = 0
17 for i in range(10):
18     a = a + 1
19     for j in range(10):
20         a = a + 1
21 print(a)

```

Don't go over this section too fast. Give it a try and predict the output of the code above. Then copy it into a Python program and run it to see if you are right. If you aren't, figure out why.

1.11.2 Example for Loops

This example code covers common for loops and shows how they work.

Listing 27: for_loop_examples.py

```

1 # Sample Python/Pygame Programs
2 # Simpson College Computer Science
3 # http://programarcadegames.com/
4 # http://simpson.edu/computer-science/
5
6 # Print 'Hi' 10 times
7 for i in range(10):
8     print("Hi")
9
10 # Print 'Hello' 5 times and 'There' once
11 for i in range(5):
12     print("Hello")
13 print("There")
14
15 # Print 'Hello' 'There' 5 times

```

(continues on next page)

(continued from previous page)

```

16 for i in range(5):
17     print("Hello")
18     print("There")
19
20 # Print the numbers 0 to 9
21 for i in range(10):
22     print(i)
23
24 # Two ways to print the numbers 1 to 10
25 for i in range(1, 11):
26     print(i)
27
28 for i in range(10):
29     print(i + 1)
30
31 # Two ways to print the even numbers 2 to 10
32 for i in range(2, 12, 2):
33     print(i)
34
35 for i in range(5):
36     print((i + 1) * 2)
37
38 # Count down from 10 down to 1 (not zero)
39 for i in range(10, 0, -1):
40     print(i)
41
42 # Print numbers out of a list
43 for i in [2, 6, 4, 2, 4, 6, 7, 4]:
44     print(i)
45
46 # What does this print? Why?
47 for i in range(3):
48     print("a")
49     for j in range(3):
50         print("b")
51
52 # What is the value of a?
53 a = 0
54 for i in range(10):
55     a = a + 1
56 print(a)
57
58 # What is the value of a?
59 a = 0
60 for i in range(10):
61     a = a + 1
62 for j in range(10):
63     a = a + 1
64 print(a)
65
66 # What is the value of a?
67 a = 0
68 for i in range(10):
69     a = a + 1
70     for j in range(10):
71         a = a + 1
72 print(a)

```

(continues on next page)

(continued from previous page)

```

73
74 # What is the value of sum?
75 sum = 0
76 for i in range(1, 101):
77     sum = sum + i

```

1.11.3 While Loops

A `for` loop is used when a program knows it needs to repeat a block of code for a certain number of times. A `while` loop is used when a program needs to loop until a particular condition occurs.

Oddly enough, a `while` loop can be used anywhere a `for` loop is used. It can be used to loop until an increment variable reaches a certain value. Why have a `for` loop if a `while` loop can do everything? The `for` loop is simpler to use and code. A `for` loop that looks like this:

Listing 28: Using a `for` loop to print the numbers 0 to 9

```

1 for i in range(10):
2     print(i)

```

Can be done with a `while` loop that looks like this:

Listing 29: Using a `while` loop to print the numbers 0 to 9

```

1 i = 0
2 while i < 10:
3     print(i)
4     i = i + 1

```

Line 1 of the `while` loop sets up a “sentinel” variable that will be used to count the number of times the loop has been executed. This happens automatically in a `for` loop eliminating one line of code. Line 2 contains the actual `while` loop. The format of the `while` loop is very similar to the `if` statement. If the condition holds, the code in the loop will repeat. Line 4 adds to the increment value. In a `for` loop this happens automatically, eliminating another line of code. As one can see from the code, the `for` loop is more compact than a `while` loop and is easier to read. Otherwise programs would do everything with a `while` loop.

A common mistake is to confuse the `for` loop and the `while` loop. The code below shows a programmer that can't quite make up his/her mind between a `for` loop or a `while` loop.

Listing 30: Example of a confused loop

```

1 while range(10):
2     print(i)

```

Don't use `range` with a `while` loop!

The `range` function only works with the `for` loop. Do not use it with the `while` loop!

Using Increment Operators

Increment operators are often used with `while` loops. It is possible to short-hand the code:

```
i = i + 1
```

With the following:

```
i += 1
```

In the while loop it would look like:

```
1 i = 0
2 while i < 10:
3     print(i)
4     i += 1
```

This can be done with subtraction and multiplication as well. For example:

```
i *= 2
```

Is the same as:

```
i = i * 2
```

See if you can figure out what would this print:

```
1 i = 1
2 while i <= 2 ** 32:
3     print(i)
4     i *= 2
```

Looping Until User Wants To Quit

A very common operation is to loop until the user performs a request to quit:

Listing 31: Looping until the user wants to quit

```
1 quit = "\n"
2 while quit == "\n":
3     quit = input("Do you want to quit? ")
```

There may be several ways for a loop to quit. Using a Boolean variable to trigger the event is a way of handling that. Here's an example:

Listing 32: Looping until the game is over or the user wants to quit

```
1 done = False
2 while not done:
3     quit = input("Do you want to quit? ")
4     if quit == "y":
5         done = True
6
7     attack = input("Does your elf attack the dragon? ")
8     if attack == "y":
9         print("Bad choice, you died.")
10        done = True
```

This isn't perfect though, because if the user says she wants to quit, the code will still ask if she wants to attack the dragon. How could you fix this?

Here is an example of using a while loop where the code repeats until the value gets close enough to zero:

```

1 value = 0
2 increment = 0.5
3 while value < 0.999:
4     value += increment
5     increment *= 0.5
6     print(value)

```

Common Problems With `while` Loops

The programmer wants to count down from 10. What is wrong and how can it be fixed?

```

1 i = 10
2 while i == 0:
3     print(i)
4     i -= 1

```

What is wrong with this loop that tries to count to 10? What will happen when it is run? How should it be fixed?

```

1 i = 1
2 while i < 10:
3     print(i)

```

1.11.4 Example `while` Loops

Here's a program that covers the different uses of the while loop that we just talked about.

Listing 33: `while_loop_examples.py`

```

# Sample Python/Pygame Programs
# Simpson College Computer Science
# http://programarcadegames.com/
# http://simpson.edu/computer-science/

# A while loop can be used anywhere a for loop is used:
i = 0
while i < 10:
    print(i)
    i = i + 1

# This is the same as:
for i in range(10):
    print(i)

# It is possible to short hand the code:
# i = i + 1
# With the following:
# i += 1
# This can be done with subtraction, and multiplication as well.
i = 0
while i < 10:
    print(i)
    i += 1

# What would this print?

```

(continues on next page)

(continued from previous page)

```

27 i = 1
28 while i <= 2**32:
29     print(i)
30     i *= 2
31
32 # A very common operation is to loop until the user performs
33 # a request to quit
34 quit = "n"
35 while quit == "n":
36     quit = input("Do you want to quit? ")
37
38 # There may be several ways for a loop to quit. Using a boolean
39 # to trigger the event is a way of handling that.
40 done = False
41 while not done:
42     quit = input("Do you want to quit? ")
43     if quit == "y":
44         done = True
45
46     attack = input("Does your elf attack the dragon? ")
47     if attack == "y":
48         print("Bad choice, you died.")
49         done = True
50
51 value = 0
52 increment = 0.5
53 while value < 0.999:
54     value += increment
55     increment *= 0.5
56     print(value)
57
58 # -- Common problems with while loops --
59
60 # The programmer wants to count down from 10
61 # What is wrong and how to fix it?
62 i = 10
63 while i == 0:
64     print(i)
65     i -= 1
66
67 # What is wrong with this loop that tries
68 # to count to 10? What will happen when it is run?
69 i = 1
70 while i < 10:
71     print(i)

```

1.11.5 The break And continue Statements

If you are in the middle of a loop, and your code encounters a `break` statement, you'll immediately exit the loop.

```

1 while True: # Loop forever
2     quit = input("Do you want to quit? ")
3     if quit == "y":
4         break
5

```

(continues on next page)

(continued from previous page)

```

6 attack = input("Does your elf attack the dragon? ")
7 if attack == "y":
8     print("Bad choice, you died.")
9     break

```

If you are in the middle of a loop, and your code encounters a `continue` statement, you'll immediately be sent back up to the top of the loop.

1.11.6 Random Numbers

Random numbers are heavily used in computer science for programs that involve games or simulations.

The `randrange` Function

By default, Python does not know how to make random numbers. It is necessary to have Python import a code library that can create random numbers. So to use random numbers, the first thing that should appear at the top of the program is an import statement:

```
import random
```

Just like with `pygame`, it is important not to create a file with the same name as what is being imported. Creating a file called `random.py` will cause Python to start importing that file instead of the system library that creates random numbers.

After this, random numbers can be created with the `randrange` function. For example, this code creates random numbers from 0 to 49. By default the lower bound is 0.

Listing 34: Random number from 0 to 49

```
my_number = random.randrange(50)
```

The next code example generates random numbers from 100 to 200. Just like the `range` function the second parameter specifies an upper-bound that is not inclusive. Therefore if you want random numbers up to and including 200, specify 201.

Listing 35: Random number from 100 to 200

```
my_number = random.randrange(100, 201)
```

Random Chance

Often in games there is a random chance of something happening. How do we program this? It isn't too hard. Here's an example where we have a 1 in 5 chance of meeting a dragon:

Listing 36: Random Chance of Something Happening

```

1 import random
2
3
4 for i in range(20):
5     if random.randrange(5) == 0:
6         print("DRAGON!!!!")

```

(continues on next page)

(continued from previous page)

```
7     else:  
8         print("No dragon.")
```

This code repeats twenty times. Inside the loop, we “roll the dice” and get a number between 0 and 4. If we roll a 0, then we encounter a dragon. Otherwise we don’t.

The random Function

All of the prior code generates integer numbers. If a floating point number is desired, a programmer may use the `random` function.

The code below generates a random number from 0 to 1 such as 0.4355991106620656.

Listing 37: Random floating point number from 0 to 1

```
my_number = random.random()
```

With some simple math, this number can be adjusted. For example, the code below generates a random floating point number between 10 and 15:

Random floating point number between 10 and 15

```
my_number = random.random() * 5 + 10
```

Number Guessing Game Example

Here is the code for the number guessing game at the start of the chapter.

```
1 """  
2     Random Number Guessing Game  
3 """  
4 import random  
5  
6  
7 def main():  
8  
9     print("Hi! I'm thinking of a random number between 1 and 100.")  
10  
11    # NEW CONCEPT  
12    # Create a secret number  
13    secret_number = random.randrange(1, 101)  
14  
15    # Initialize our attempt count, we start with attempt 1.  
16    user_attempt_number = 1  
17  
18    # Set user guess to something secret number can't be, so we can  
19    # get our 'while' loop started.  
20    user_guess = 0  
21  
22    # NEW CONCEPT  
23    # Loop until user_guess our secret number, or we run out of attempts.  
24    while user_guess != secret_number and user_attempt_number < 8:  
25  
26        # Tell the user what attempt we are on, and get their guess:  
27  
28        user_guess = int(input("What's your guess? "))  
29  
30        user_attempt_number += 1  
31  
32        if user_guess < secret_number:  
33            print("Too low!")  
34        elif user_guess > secret_number:  
35            print("Too high!")  
36        else:  
37            print("You got it!")  
38  
39    if user_attempt_number == 8:  
40        print("Sorry, you're out of attempts!")  
41  
42    else:  
43        print("You got it in", user_attempt_number, "attempts!")  
44  
45    print("The secret number was", secret_number)
```

(continues on next page)

(continued from previous page)

```

27     print("--- Attempt", user_attempt_number)
28     user_input_text = input("Guess what number I am thinking of: ")
29     user_guess = int(user_input_text)
30
31     # Print if we are too high or low, or we got it.
32     if user_guess > secret_number:
33         print("Too high.")
34     elif user_guess < secret_number:
35         print("Too low.")
36     else:
37         print("You got it!")
38
39     # Add to the attempt count
40     user_attempt_number += 1
41
42     # Here, check to see if the user didn't guess the answer, and ran out of tries.
43     # Let her know what the number was, so she doesn't spend the rest of her life
44     # wondering.
45     if user_guess != secret_number:
46         print("Aw, you ran out of tries. The number was " + str(secret_number) + ".")
47
48 # Call the main function
49 main()

```

Mudball Example

This is a fun text-only game that two players can play. It uses a few concepts we haven't covered yet.

```

1 """
2 This is a sample text-only game that demonstrates the use of functions.
3 The game is called "Mudball" and the players take turns lobbing mudballs
4 at each other until someone gets hit.
5 """
6
7 import math
8 import random
9
10
11 def print_instructions():
12     """ This function prints the instructions. """
13
14     # You can use the triple-quote string in a print statement to
15     # print multiple lines.
16     print("""
17 Welcome to Mudball! The idea is to hit the other player with a mudball.
18 Enter your angle (in degrees) and the amount of PSI to charge your gun
19 with.
20     """)
21
22
23 def calculate_distance(psi, angle_in_degrees):
24     """ Calculate the distance the mudball flies. """
25     angle_in_radians = math.radians(angle_in_degrees)
26     distance = .5 * psi ** 2 * math.sin(angle_in_radians) * math.cos(angle_in_radians)
27     return distance

```

(continues on next page)

(continued from previous page)

```

28
29
30 def get_user_input(name):
31     """ Get the user input for psi and angle. Return as a list of two
32     numbers. """
33     # Later on in the 'exceptions' chapter, we will learn how to modify
34     # this code to not crash the game if the user types in something that
35     # isn't a valid number.
36     psi = float(input(name + " charge the gun with how many psi? "))
37     angle = float(input(name + " move the gun at what angle? "))
38     return psi, angle
39
40
41 def get_player_names():
42     """ Get a list of names from the players. """
43     print("Enter player names. Enter as many players as you like.")
44     done = False
45     players = []
46     while not done:
47         player = input("Enter player (hit enter to quit): ")
48         if len(player) > 0:
49             players.append(player)
50         else:
51             done = True
52
53     print()
54     return players
55
56
57 def process_player_turn(player_name, distance_apart):
58     """ The code runs the turn for each player.
59     If it returns False, keep going with the game.
60     If it returns True, someone has won, so stop. """
61     psi, angle = get_user_input(player_name)
62
63     distance_mudball = calculate_distance(psi, angle)
64     difference = distance_mudball - distance_apart
65
66     # By looking ahead to the chapter on print formatting, these
67     # lines could be made to print the numbers in a nice formatted
68     # manner.
69     if difference > 1:
70         print("You went", difference, "yards too far!")
71     elif difference < -1:
72         print("You were", difference * -1, "yards too short!")
73     else:
74         print("Hit!", player_name, "wins!")
75         return True
76
77     print()
78     return False
79
80
81 def main():
82     """ Main program. """
83
84     # Get the game started.

```

(continues on next page)

(continued from previous page)

```

85     print_instructions()
86     player_names = get_player_names()
87     distance_apart = random.randrange(50, 150)
88
89     # Keep looking until someone wins
90     done = False
91     while not done:
92         # Loop for each player
93         for player_name in player_names:
94             # Process their turn
95             done = process_player_turn(player_name, distance_apart)
96             # If someone won, 'break' out of this loop and end the game.
97             if done:
98                 break
99
100    if __name__ == "__main__":
101        main()

```

1.12 Advanced Looping



Games involve a lot of complex loops. This is a “challenge chapter” to learn how to be an expert with loops. If you can understand the problems in this chapter, by the end of it you can certify yourself as a loop expert.

If becoming a loop expert isn’t your goal, at least make sure you can write out the answers for the first eight problems. That will give you enough knowledge to continue this book. (Besides, being a loop expert never got anyone a date. Except for that guy in the [Groundhog Day](#) movie.)

There are video explanations for the answers on-line.

1.12.1 Print Statement End Characters

By default, the “print” statement puts a *carriage return* at the end of what is printed out. As we explained back in the first chapter, the carriage return is a character that moves the next line of output to be printed to the next line. Most of the time this is what we want. Sometimes it isn’t. If we want to continue printing on the same line, we can change the default character printed at the end. This is an example before we change the ending character:

```

print("Pink")
print("Octopus")

```

...which will print out:

```

Pink
Octopus

```

But if we wanted the code output to print on the same line, it can be done by using a new option to set the “end” character. For example:

```
print("Pink", end="")
print("Octopus")
```

This will print:

```
PinkOctopus
```

We can also use a space instead of setting it to an empty string:

```
print("Pink", end=" ")
print("Octopus")
```

This will print:

```
Pink Octopus
```

Here's another example, using a variable:

```
i = 0
print(i, end=" ")
i = 1
print(i, end=" ")
```

This will print:

```
0 1
```

1.12.2 Advanced Looping Problems

Problem 1

Write code that will print ten asterisks (*) like the following:

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

Have this code print using a “for” loop, and print each asterisk individually, rather than just printing ten asterisks with one “print” statement. This can be done in two lines of code, a “for” loop and a “print” statement. When you have figured it out, or given up, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/problem_1.php

Problem 2

Write code that will print the following:

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

This is just like the prior problem, but also printing five and twenty stars. Copy and paste from the prior problem, adjusting the “for” loop as needed.

When you have figured it out, or given up, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/problem_2.php

Problem 3

Use two “for” loops, one of them nested inside the other, to print the following 10x10 rectangle:

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

To start, take a look at Problem 1. The code in Problem 1 generates one line of asterisks. It needs to be repeated ten times. Work on this problem for at least ten minutes before looking at the answer.

When you have figured it out, or given up, here it is:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/10x10box.php

Problem 4

Use two “for” loops, one of them nested, to print the following 5x10 rectangle:

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

This is a lot like the prior problem. Experiment with the ranges on the loops to find exactly what the numbers passed to the “range” function control.

When you have figured it out, or given up, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/problem_4.php

Problem 5

Use two “for” loops, one of them nested, to print the following 20x5 rectangle:

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

Again, like Problem 3 and Problem 4, but with different range values.

When you have figured it out, or given up, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/problem_5.php

Problem 6

Write code that will print the following:

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

Use two nested loops. Print the first line with a loop, and not with:

```
print("0 1 2 3 4 5 6 7 8 9")
```

Attention: First, create a loop that prints the first line. Then enclose it in another loop that repeats the line 10 times. Use either “i” or “j” variables for what the program prints. This example and the next one helps reinforce what those index variables are doing.

Work on this problem for at least ten minutes before looking at the answer. The process of spending ten minutes working on the answer is far more important than the answer itself.

http://ProgramArcadeGames.com/chapters/06_back_to_looping/number_square_answer.php

Problem 7

Adjust the prior program to print:

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

Answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/problem_7.php

Problem 8

Write code that will print the following:

```
0  
0 1  
0 1 2
```

(continues on next page)

(continued from previous page)

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

Tip: This is just problem 6, but the inside loop no longer loops a fixed number of times. Don't use `range(10)`, but adjust that range amount.

After working at least ten minutes on the problem, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/top_right_triangle.php

Make sure you can write out the code for this and the prior problems. Yes, this practice is work, but it will pay off later and you'll save time in the long run.

Problem 9

Write code that will print the following:

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

This one is difficult. Tip: Two loops are needed inside the outer loop that controls each row. First, a loop prints spaces, then a loop prints the numbers. Loop both these for each row. To start with, try writing just one inside loop that prints:

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

Then once that is working, add a loop after the outside loop starts and before the already existing inside loop. Use this new loop to print enough spaces to right justify the other loops.

After working at least ten minutes on the problem, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/bottom_left_triangle.php

Problem 10

Write code that will print the following (Getting the alignment is hard, at least get the numbers):

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Tip: Start by adjusting the code in problem 1 to print:

0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8
0	2	4	6	8	10	12	14	16
0	3	6	9	12	15	18	21	24
0	4	8	12	16	20	24	28	32
0	5	10	15	20	25	30	35	40
0	6	12	18	24	30	36	42	48
0	7	14	21	28	35	42	49	56
0	8	16	24	32	40	48	56	64
0	9	18	27	36	45	54	63	72

Then adjust the code to print:

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Finally, use an “if” to print spaces if the number being printed is less than 10.

After working at least ten minutes on the problem, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/multiplication_table.php

Problem 11

Write code that will print the following:

1	2	1														
1	2	3	2	1												
1	2	3	4	3	2	1										
1	2	3	4	5	4	3	2	1								
1	2	3	4	5	6	5	4	3	2	1						
1	2	3	4	5	6	7	6	5	4	3	2	1				
1	2	3	4	5	6	7	8	7	6	5	4	3	2	1		
1	2	3	4	5	6	7	8	9	8	7	6	5	4	3	2	1

Tip: first write code to print:

```

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8 9

```

Then write code to print:

```

1
1 2 1
1 2 3 2 1
1 2 3 4 3 2 1
1 2 3 4 5 4 3 2 1
1 2 3 4 5 6 5 4 3 2 1
1 2 3 4 5 6 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1

```

Then finish by adding spaces to print the final answer.

After working at least ten minutes on the problem, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/top_triangle.php

Problem 12

Write code that will print the following:

```

      1
      1 2 1
      1 2 3 2 1
      1 2 3 4 3 2 1
      1 2 3 4 5 4 3 2 1
      1 2 3 4 5 6 5 4 3 2 1
      1 2 3 4 5 6 7 6 5 4 3 2 1
      1 2 3 4 5 6 7 8 7 6 5 4 3 2 1
      1 2 3 4 5 6 7 8 9 8 7 6 5 4 3 2 1
      1 2 3 4 5 6 7
      1 2 3 4 5 6
      1 2 3 4 5
      1 2 3 4
      1 2 3
      1 2
      1

```

This can be done by combining problems 11 and 9.

After working at least ten minutes on the problem, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/three_quarters.php

Problem 13

Write code that will print the following:

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

After working at least ten minutes on the problem, here is the answer:

http://ProgramArcadeGames.com/chapters/06_back_to_looping/full_diamond.php

1.13 Introduction to Lists

1.13.1 Data Types

So far this book has shown four types of data:

- String (a string is short for “string of characters,” which normal people think of as text.)
- Integer
- Floating point
- Boolean

Python can display what type of data a value is with the `type` function. For example, what type of data is a 3?

```
# Print the type of data a 3 is:
print(type(3))
```

```
<class 'int'>
```

This `type` function isn’t useful for other programming in this book, but it is good to demonstrate the types of data introduced so far.

Here we set `x` to the four types of data we’ve used so far, and call the `type` command to see how Python classifies that data:

```
x = 3
print("x =", x, "and is of type:", type(x))

x = 3.145
```

(continues on next page)

(continued from previous page)

```
print("x =", x, "and is of type:", type(x))

x = "Hi there"
print("x =", x, "and is of type:", type(x))

x = True
print("x =", x, "and is of type:", type(x))
```

The output:

```
x = 3 and is of type: <class 'int'>
x = 3.145 and is of type: <class 'float'>
x = Hi there and is of type: <class 'str'>
x = True and is of type: <class 'bool'>
```

Note: More than one coin to collect? Use a list!

The two new types of data introduced in this chapter are Lists and Tuples. Lists are similar to another data structure called an array. A list can be resized, but an array can not. A course in data structures will teach you the details, but it that is beyond the scope of this book. Try running the following commands in the interactive Python shell and see what is displayed:

```
x = (2, 3, 4, 5)
print("x =", x, "and is of type:", type(x))

x = [2, 3, 4, 5]
print("x =", x, "and is of type:", type(x))
```

The output:

```
x = (2, 3, 4, 5) and is of type: <class 'tuple'>
x = [2, 3, 4, 5] and is of type: <class 'list'>
```

1.13.2 Working With Lists

You've created grocery lists, to-do lists, bucket lists, but how do you create a list on the computer?

To create a list and print it out, try the following:

```
x = [10, 20]
print(x)
```

The output:

```
[10, 20]
```

To print an individual element in a list:

```
print(x[0])
```

The output:

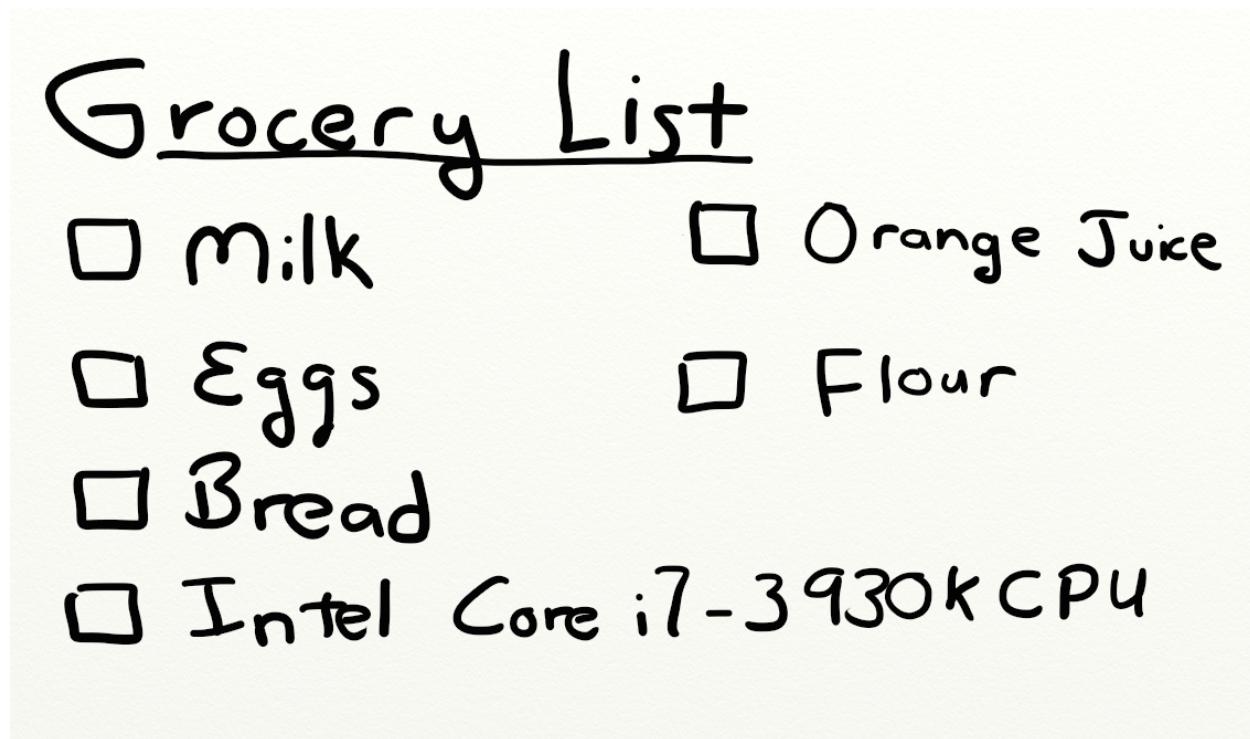


Fig. 11: Even computers use lists

10

This number with the item's location is called the index. Note that list locations start at zero. So a list or array with 10 elements does not have an element in spot [10]. Just spots [0] through [9]. It can be very confusing to create an list of 10 items and then not have an item 10, but most computer languages start counting at 0 rather than 1.

Think of a list as an ice cube tray that holds numbers, as shown in Figure 7.2. The values are stored inside each tray spot, and written on the side of the tray are numbers starting at zero that identify the location of each spot.

Attention: Don't mix the index and the value!

Remember, there are two sets of numbers to consider when working with a list of numbers: the position and the value. The position, also known as index, refers to where a value is. The value is the actual number stored at that location. When working with a list or array, make sure to think if you need the location or the value.

It is easy to get the value given the location, but it is harder to get the location given the value. Chapter 15 is dedicated to answering how to find the location of a particular value.

You can also access elements from the back-side of an array using negative numbers. (Not all languages support this.) For example:

```
x = [10, 20, 30]
print(x[-1])
```

The output:

30

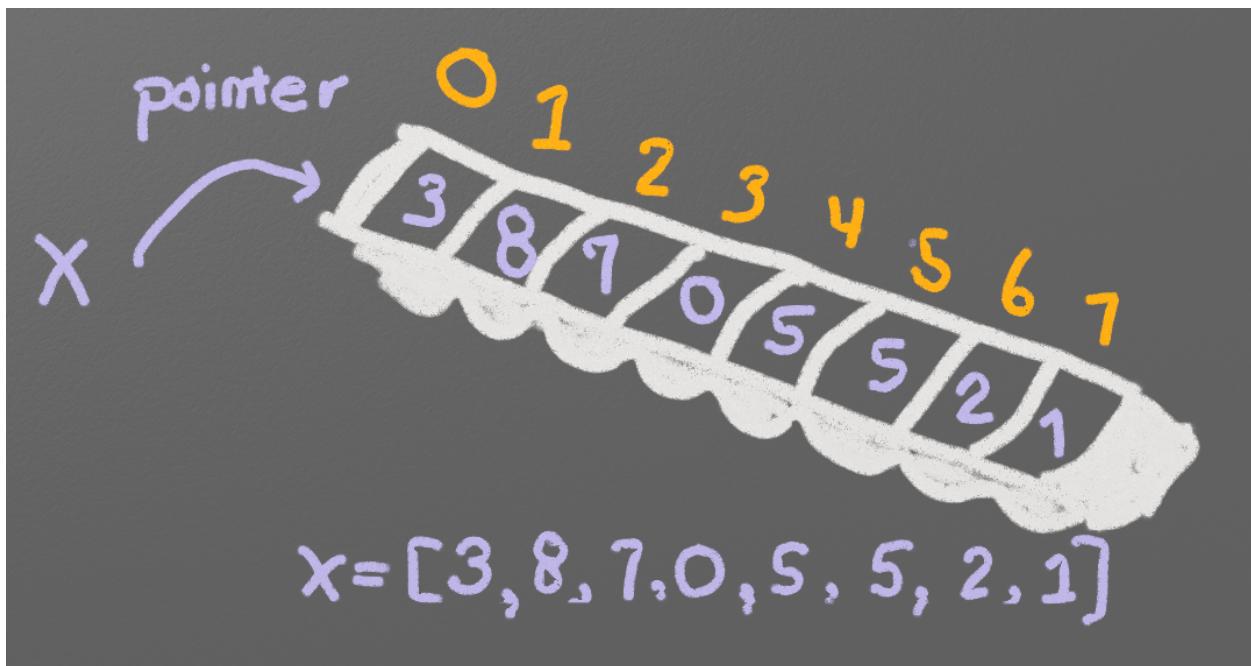


Fig. 12: Lists are like ice cube trays

A program can assign new values to an individual element in a list. In the case below, the first spot at location zero (not one) is assigned the number 22.

```
x = [1, 2]
print(x)

x[0] = 22
print(x)
```

```
[1, 2]
[22, 2]
```

Also, a program can create a “tuple.” This data type works just like a list, but with two differences. First, it is created with parentheses rather than square brackets. Second, it is not possible to change the tuple once created. See below:

```
x = (1, 2)
print(x)

x[0] = 22
print(x)
```

```
[1, 2]
Traceback (most recent call last):
  File "<pyshell#18>", line 4, in <module>
    x[0] = 22
TypeError: 'tuple' object does not support item assignment
```

As can be seen from the output of the code above, we can't assign an item in the tuple a new value. Why would we want this limitation? First, the computer can run faster if it knows the value won't change. Second, some lists we don't want to change, such as a list of RGB colors for red. The color red doesn't change, therefore an immutable tuple is a better choice.

1.13.3 Create an Empty List

Occasionally we need to create a list that is empty. We'll use this in a bit when we start with an empty list and build on it. How do I create an empty list? Easy:

```
# Create an empty list
my_list = []
```

1.13.4 Iterating (Looping) Through a List

If a program needs to iterate through each item in a list, such as to print it out, there are two types of for loops that can do this.

The first method to iterate through each item in a loop is by using a “for-each” loop. This type of loop takes a collection of items, and loops the code once per item. It will take a copy of the item and store it in a variable for processing.

The format of the command:

```
for item_variable in list_name:
```

Here are some examples:

```
my_list = [101, 20, 10, 50, 60]
for item in my_list:
    print(item)
```

```
101
20
10
50
60
```

Programs can store strings in lists too:

```
my_list = ["Spoon", "Fork", "Knife"]
for item in my_list:
    print(item)
```

```
Spoon
Knife
Fork
```

Lists can even contain other lists. This iterates through each item in the main list, but not in sublists.

```
my_list = [[2, 3], [4, 3], [6, 7]]
for item in my_list:
    print(item)
```

```
[2,3]
[4,3]
[6,7]
```

The other way to iterate through a list is to use an index variable and directly access the list rather than through a copy of each item. To use an index variable, the program counts from 0 up to the length of the list. If there are ten elements, the loop must go from 0 to 9 for a total of ten elements.

The length of a list may be found by using the `len` function. Combining that with the `range` function allows the program to loop through the entire list.

```
my_list = [101, 20, 10, 50, 60]
for index in range(len(my_list)):
    print(my_list[index])
```

```
101
20
10
50
60
```

This method is more complex, but is also more powerful. Because we are working directly with the list elements, rather than a copy, the list can be modified. The for-each loop does not allow modification of the original list.

1.13.5 Looping With Both An Index And Element

If you want both the index, like a `for i in range` gives you, and the element, like a `for item in my_list` gives you, the proper Python-ic way to use the `enumerate` function like this:

```
for index, value in enumerate(my_list):
    print(index, value)
```

1.13.6 Adding to a List

New items may be added to a list (but not a tuple) by using the `append` command. For example:

```
my_list = [2, 4, 5, 6]
print(my_list)
my_list.append(9)
print(my_list)
```

```
[2, 4, 5, 6]
[2, 4, 5, 6, 9]
```

Side note: If performance while appending is a concern, it is very important to understand how a list is being implemented. For example, if a list is implemented as an *array data type*, then appending an item to the list is a lot like adding a new egg to a full egg carton. A new egg carton must be built with thirteen spots. Then twelve eggs are moved over. Then the thirteenth egg is added. Finally the old egg carton is recycled. Because this can happen behind the scenes in a function, programmers may forget this and let the computer do all the work. It would be more efficient to simply tell the computer to make an egg carton with enough spots to begin with. Thankfully, Python does not implement a list as an array data type. But it is important to pay attention to your next semester data structures class and learn how all of this works.

To create a list from scratch, it is necessary to create a blank list and then use the `append` function to build it based upon user input:

Listing 38: Creating a list of numbers from user input

```
# Create an empty list
my_list = []
```

(continues on next page)

(continued from previous page)

```

for i in range(5):
    user_input = input( "Enter an integer: ")
    user_input = int(user_input)
    my_list.append(user_input)
    print(my_list)

```

```

Enter an integer: 4
[4]
Enter an integer: 5
[4, 5]
Enter an integer: 3
[4, 5, 3]
Enter an integer: 1
[4, 5, 3, 1]
Enter an integer: 8
[4, 5, 3, 1, 8]

```

If a program needs to create an array of a specific length, all with the same value, a simple trick is to use the following code:

Listing 39: Create an array with 100 zeros

```

1 # Create an array with 100 zeros.
2 my_list = [0] * 100

```

1.13.7 Summing or Modifying a List

Creating a running total of an array is a common operation. Here's how it is done:

Listing 40: Summing the values in a list v1

```

1 # Copy of the array to sum
2 my_list = [5, 76, 8, 5, 3, 3, 56, 5, 23]
3
4 # Initial sum should be zero
5 list_total = 0
6
7 # Loop from 0 up to the number of elements
8 # in the array:
9 for index in range(len(my_list)):
10     # Add element 0, next 1, then 2, etc.
11     list_total += my_list[index]
12
13 # Print the result
14 print(list_total)

```

The same thing can be done by using a `for` loop to iterate the array, rather than count through a range:

Listing 41: Summing the values in a list v2

```

1 # Copy of the array to sum
2 my_list = [5, 76, 8, 5, 3, 3, 56, 5, 23]
3
4 # Initial sum should be zero

```

(continues on next page)

(continued from previous page)

```

5 list_total = 0
6
7 # Loop through array, copying each item in the array into
8 # the variable named item.
9 for item in my_list:
10     # Add each item
11     list_total += item
12
13 # Print the result
14 print(list_total)

```

Numbers in an array can also be changed by using a `for` loop:

Listing 42: Doubling all the numbers in a list

```

1 # Copy of the array to modify
2 my_list = [5, 76, 8, 5, 3, 3, 56, 5, 23]
3
4 # Loop from 0 up to the number of elements
5 # in the array:
6 for index in range(len(my_list)):
7     # Modify the element by doubling it
8     my_list[index] = my_list[index] * 2
9
10 # Print the result
11 print(my_list)

```

However version 2 does not work at doubling the values in an array. Why? Because `item` is a *copy* of an element in the array. The code below doubles the copy, not the original array element.

Listing 43: Bad code that doesn't double all the numbers in a list

```

1 # Copy of the array to modify
2 my_list = [5, 76, 8, 5, 3, 3, 56, 5, 23]
3
4 # Loop through each element in myArray
5 for item in my_list:
6     # This doubles item, but does not change the array
7     # because item is a copy of a single element.
8     item = item * 2
9
10 # Print the result
11 print(my_list)

```

1.13.8 Slicing Strings

Strings are actually lists of characters. They can be treated like lists with each letter a separate item. Run the following code with both versions of `x`:

Listing 44: Accessing a string as a list

```

1 x = "This is a sample string"
2 #x = "0123456789"
3
4 print("x=", x)

```

(continues on next page)

(continued from previous page)

```

5   # Accessing the first character ("T")
6   print("x[0]=", x[0])
7
8
9   # Accessing the second character ("h")
10  print("x[1]=", x[1])
11
12  # Accessing from the right side ("g")
13  print("x[-1]=", x[-1])
14
15  # Access 0-5 ("This ")
16  print("x[:6]=", x[:6])
17  # Access 6 to the end ("is a sample string")
18  print("x[6:]=", x[6:])
19  # Access 6-8
20  print("x[6:9]=", x[6:9])

```

Strings in Python may be used with some of the mathematical operators. Try the following code and see what Python does:

Listing 45: Adding and multiplying strings

```

1 a = "Hi"
2 b = "There"
3 c = "!"
4 print(a + b)
5 print(a + b + c)
6 print(3 * a)
7 print(a * 3)
8 print((a * 2) + (b * 2))

```

It is possible to get a length of a string. It is also possible to do this with any type of array.

Listing 46: Getting the length of a string or list

```

1 a = "Hi There"
2 print(len(a))
3
4 b = [3, 4, 5, 6, 76, 4, 3, 3]
5 print(len(b))

```

Since a string is an array, a program can iterate through each character element just like an array:

```

for character in "This is a test.":
    print(character)

```

Exercise: Starting with the following code:

```

1 months = "JanFebMarAprMayJunJulAugSepOctNovDec"
2 n = int(input("Enter a month number: "))

```

Print the three month abbreviation for the month number that the user enters. (Calculate the start position in the string, then use the info we just learned to print out the correct substring.)

1.13.9 Secret Codes

This code prints out every letter of a string individually:

```

1 plain_text = "This is a test. ABC abc"
2
3 for c in plain_text:
4     print(c, end=" ")

```

Computers do not actually store letters of a string in memory; computers store a series of numbers. Each number represents a letter. The system that computers use to translate numbers to letters is called *Unicode*. The full name for the encoding is Universal Character Set Transformation Format 8-bit, usually abbreviated UTF-8.

The Unicode chart covers the Western alphabet using the numbers 0-127. Each Western letter is represented by one byte of memory. Other alphabets, like Cyrillic, can take multiple bytes to represent each letter. A partial copy of the Unicode chart is below:

Value	Character	Value	Character	Value	Character	Value	Character
40	(61	=	82	R	103	g
41)	62	>	83	S	104	h
42	•	63	?	84	T	105	i
43	•	64	@	85	U	106	j
44	,	65	A	86	V	107	k
45	•	66	B	87	W	108	l
46	.	67	C	88	X	109	m
47	/	68	D	89	Y	110	n
48	0	69	E	90	Z	111	o
49	1	70	F	91	[112	p
50	2	71	G	92		113	q
51	3	72	H	93]	114	r
52	4	73	I	94	^	115	s
53	5	74	J	95	_	116	t
54	6	75	K	96	‘	117	u
55	7	76	L	97	a	118	v
56	8	77	M	98	b	119	w
57	9	78	N	99	c	120	x
58	:	79	O	100	d	121	y
59	;	80	P	101	e	122	z
60	<	81	Q	102	f		

For more information about ASCII (which has the same values as Unicode for the Western alphabet) see:

<http://en.wikipedia.org/wiki/ASCII>

For a video that explains the beauty of Unicode, see here:

<http://hackaday.com/2013/09/27/utf-8-the-most-elegant-hack>

This next set of code converts each of the letters in the prior example to its ordinal value using UTF-8:

```
plain_text = "This is a test. ABC abc"

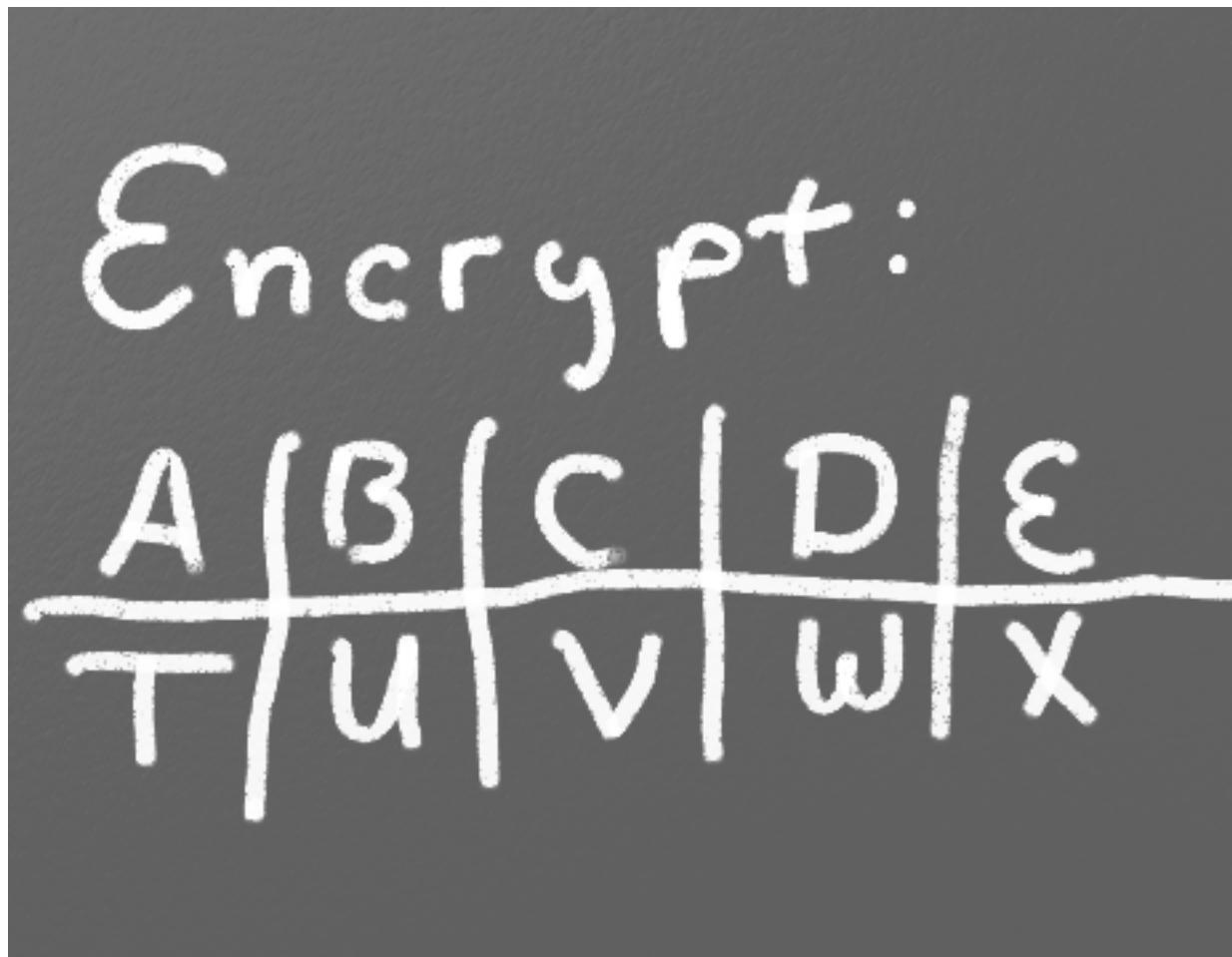
for c in plain_text:
    print(ord(c), end=" ")
```

This next program takes each UTF-8 value and adds one to it. Then it prints the new UTF-8 value, then converts the value back to a letter.

```
plain_text = "This is a test. ABC abc"

for c in plain_text:
    x = ord(c)
    x = x + 1
    c2 = chr(x)
    print(c2, end="")
```

The next code listing takes each UTF-8 value and adds one to it, then converts the value back to a letter.



Listing 47: simple_encryption.py

```
1 # Sample Python/Pygame Programs
2 # Simpson College Computer Science
3 # http://programarcadegames.com/
4 # http://simpson.edu/computer-science/
```

(continues on next page)

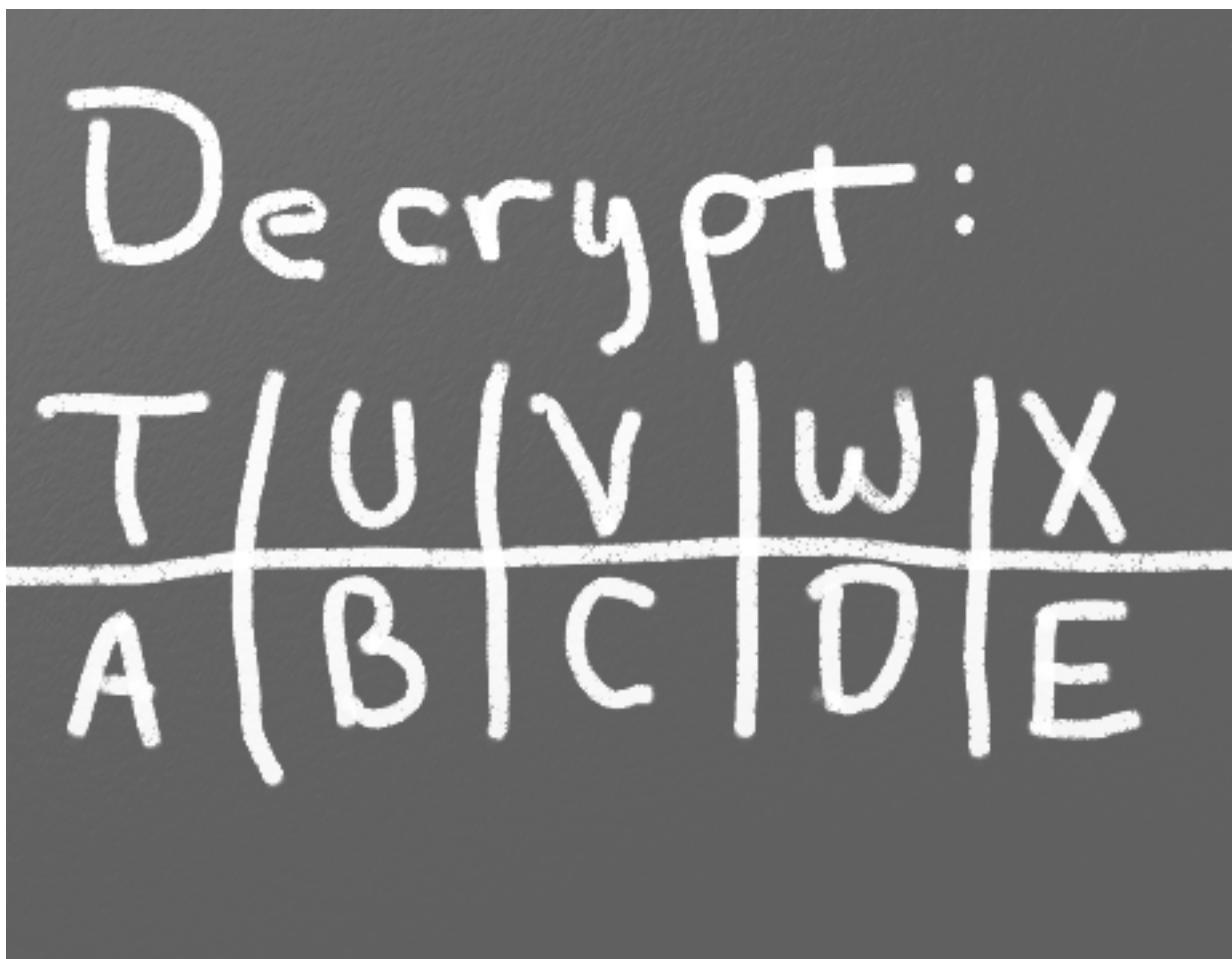
(continued from previous page)

```

5 # Explanation video: http://youtu.be/sxFIxD8Gd3A
6
7 plain_text = "This is a test. ABC abc"
8
9 encrypted_text = ""
10 for c in plain_text:
11     x = ord(c)
12     x = x + 1
13     c2 = chr(x)
14     encrypted_text = encrypted_text + c2
15
16 print(encrypted_text)

```

Finally, the last code takes each UTF-8 value and subtracts one from it, then converts the value back to a letter. By feeding this program the output of the previous program, it serves as a decoder for text encoded by the prior example.



Listing 48: simple_decryption.py

```

1 # Sample Python/Pygame Programs
2 # Simpson College Computer Science
3 # http://programarcadegames.com/
4 # http://simpson.edu/computer-science/
5

```

(continues on next page)

(continued from previous page)

```
6 # Explanation video: http://youtu.be/sxFIxD8Gd3A
7
8 encrypted_text = "Uijt!jt!b!uftu!/BCD!bcd"
9
10 plain_text = ""
11 for c in encrypted_text:
12     x = ord(c)
13     x = x - 1
14     c2 = chr(x)
15     plain_text = plain_text + c2
16 print(plain_text)
```

1.13.10 Associative Arrays

Python is not limited to using numbers as an array index. It is also possible to use an associative array. An associative array works like this:

```
1 # Create an empty associative array
2 # (Note the curly braces.)
3 x = {}
4
5 # Add some stuff to it
6 x["fred"] = 2
7 x["scooby"] = 8
8 x["wilma"] = 1
9
10 # Fetch and print an item
11 print(x["fred"])
```

You won't really need associative arrays for this class, but I think it is important to point out that it is possible.

1.14 Introduction to Classes

Classes and objects are very powerful programming tools. They make programming easier. In fact, you are already familiar with the concept of classes and objects. A class is a “classification” of an object. Like “person” or “image.” An object is a particular instance of a class. Like “Mary” is an instance of “Person.”

Objects have attributes, such as a person’s name, height, and age. Objects also have methods. Methods define what an object can do, like run, jump, or sit.

1.14.1 Why Learn About Classes?

Each character in an adventure game needs data: a name, location, strength, are they raising their arm, what direction they are headed, etc. Plus those characters do things. They run, jump, hit, and talk.

Without classes, our Python code to store this data might look like:

```
1 name = "Link"
2 outfit = "Green"
3 max_hit_points = 50
4 current_hit_points = 50
```

In order to do anything with this character, we'll need to pass that data to a function:

```
1 def display_character(name, sex, max_hit_points, current_hit_points):
2     print(name, sex, max_hit_points, current_hit_points)
```

Now imagine creating a program that has a set of variables like that for each character, monster, and item in our game. Then we need to create functions that work with those items. We've now waded into a quagmire of data. All of a sudden this doesn't sound like fun at all.

But wait, it gets worse! As our game expands, we may need to add new fields to describe our character. In this case we've added `max_speed`:

```
1 name = "Link"
2 outfit = "Green"
3 max_hit_points = 50
4 current_hit_points = 50
5 max_speed = 10
6
7 def display_character(name, outfit, max_hit_points, current_hit_points, max_speed):
8     print(name, sex, max_hit_points, current_hit_points)
```

In example above, there is only one function. But in a large video game, we might have hundreds of functions that deal with the main character. Adding a new field to help describe what a character has and can do would require us to go through each one of those functions and add it to the parameter list. That would be a lot of work. And perhaps we need to add `max_speed` to different types of characters like monsters. There needs to be a better way. Somehow our program needs to package up those data fields so they can be managed easily.

1.14.2 Defining and Creating Simple Classes

A better way to manage multiple data attributes is to *define* a structure that has all of the information. Then we can give that “grouping” of information a name, like *Character* or *Address*. This can be easily done in Python and any other modern language by using a *class*.

For example, we can *define* a class representing a character in a game:

```
1 class Character():
2     """ This is a class that represents the main character in a game. """
3     def __init__(self):
4         """ This is a method that sets up the variables in the object. """
5         self.name = ""
6         outfit = "Green"
7         self.max_hit_points = 0
8         self.current_hit_points = 0
9         self.max_speed = 0
10        self.armor_amount = 0
```

Here's another example, we *define* a class to hold all the fields for an address:

Listing 49: Define an address class

```
1 class Address():
2     """ Hold all the fields for a mailing address. """
3     def __init__(self):
4         """ Set up the address fields. """
5         self.name = ""
6         self.line1 = ""
```

(continues on next page)

(continued from previous page)

```

7     self.line2 = ""
8     self.city = ""
9     self.state = ""
10    self.zip = ""

```

In the code above, `Address` is the class name. The variables in the class, such as `name` and `city`, are called *attributes* or *fields*. (Note the similarities and differences between declaring a class and declaring a function.)

Unlike functions and variables, class names should begin with an upper case letter. While it is possible to begin a class with a lower case letter, it is not considered good practice.

The `def __init__(self):` in a special function called a *constructor* that is run automatically when the class is created. We'll discuss the constructor more in a bit.

The `self.` is kind of like the pronoun *my*. When inside the class `Address` we are talking about *my name*, *my city*, etc. We don't want to use `self.` outside of the class definition for `Address`, to refer to an `Address` field. Why? Because just like the pronoun "my," it means someone totally different when said by a different person!

To better visualize classes and how they relate, programmers often make diagrams. A diagram for the `Address` class would look like the figure below. See how the class name is on top with the name of each attribute listed below. To the right of each attribute is the data type, such as string or integer.

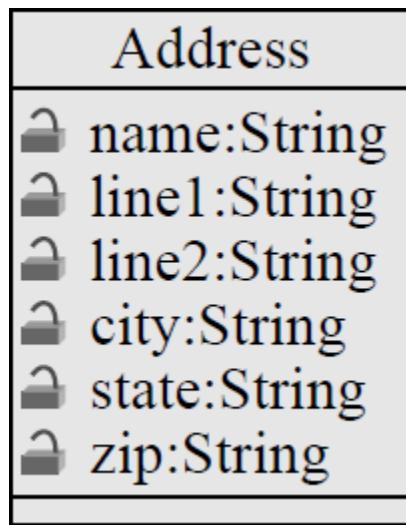


Fig. 13: Class Diagram

The class code *defines* a class but it does not actually create an *instance* of one. The code told the computer what fields an address has and what the initial default values will be. We don't actually have an address yet though. We can define a class without creating one just like we can define a function without calling it. To create a class and set the fields, look at the example below:

Listing 50: Create an instance of the address class

```

1 # Create an address
2 home_address = Address()
3
4 # Set the fields in the address
5 home_address.name = "John Smith"
6 home_address.line1 = "701 N. C Street"
7 home_address.line2 = "Carver Science Building"

```

(continues on next page)

(continued from previous page)

```

8 home_address.city = "Indianola"
9 home_address.state = "IA"
10 home_address.zip = "50125"

```

An instance of the address class is created in line 2. Note how the class `Address` name is used, followed by parentheses. The variable name can be anything that follows normal naming rules.

To set the fields in the class, a program must use the dot operator. This operator is the period that is between the `home_address` and the field name. See how lines 5-10 use the dot operator to set each field value.

A very common mistake when working with classes is to forget to specify which instance of the class you want to work with. If only one address is created, it is natural to assume the computer will know to use that address you are talking about. This is not the case however. See the example below:

```

1 class Address():
2     def __init__(self):
3         self.name = ""
4         self.line1 = ""
5         self.line2 = ""
6         self.city = ""
7         self.state = ""
8         self.zip = ""
9
10    def main():
11        # Create an address
12        my_address = Address()
13
14        # Alert! This does not set the address's name!
15        name = "Dr. Craven"
16
17        # This doesn't set the name for the address either
18        Address.name = "Dr. Craven"
19
20        # This does work:
21        my_address.name = "Dr. Craven"
22
23    main()

```

A second address can be created and fields from both instances may be used. See the example below:

Listing 51: Working with two instances of address

```

1 class Address():
2     def __init__(self):
3         self.name = ""
4         self.line1 = ""
5         self.line2 = ""
6         self.city = ""
7         self.state = ""
8         self.zip = ""
9
10    def main():
11        # Create an address
12        home_address = Address()
13
14        # Set the fields in the address

```

(continues on next page)

(continued from previous page)

```

16     home_address.name = "John Smith"
17     home_address.line1 = "701 N. C Street"
18     home_address.line2 = "Carver Science Building"
19     home_address.city = "Indianola"
20     home_address.state = "IA"
21     home_address.zip = "50125"
22
23     # Create another address
24     vacation_home_address = Address()
25
26     # Set the fields in the address
27     vacation_home_address.name = "John Smith"
28     vacation_home_address.line1 = "1122 Main Street"
29     vacation_home_address.line2 = ""
30     vacation_home_address.city = "Panama City Beach"
31     vacation_home_address.state = "FL"
32     vacation_home_address.zip = "32407"
33
34     print("The client's main home is in " + home_address.city)
35     print("His vacation home is in " + vacation_home_address.city)
36
37
38 main()

```

Line 11 creates the first instance of `Address`; line 22 creates the second instance. The variable `home_address` points to the first instance and `vacation_home_address` points to the second.

Lines 25-30 set the fields in this new class instance. Line 32 prints the city for the home address, because `home_address` appears before the dot operator. Line 33 prints the vacation address because `vacation_home_address` appears before the dot operator.

In the example `Address` is called the class because it defines a new classification for a data object. The variables `home_address` and `vacation_home_address` refer to objects because they refer to actual instances of the class `Address`. A simple definition of an object is that it is an instance of a class. Like “Bob” and “Nancy” are instances of a `Human` class.

By using www.pythontutor.com we can [visualize the execution of the code](#) (see below). There are three variables in play. One points to the class definition of `Address`. The other two variables point to the different address objects and their data.

Putting lots of data fields into a class makes it easy to pass data in and out of a function. In the code below, the function takes in an address as a parameter and prints it out on the screen. It is not necessary to pass parameters for each field of the address.

Listing 52: Working with two instances of address

```

1 # Print an address to the screen
2 def print_address(address):
3     print(address.name)
4     # If there is a line1 in the address, print it
5     if len(address.line1) > 0:
6         print(address.line1)
7     # If there is a line2 in the address, print it
8     if len(address.line2) > 0:
9         print( address.line2 )
10    print(address.city + ", " + address.state + " " + address.zip)
11

```

(continues on next page)

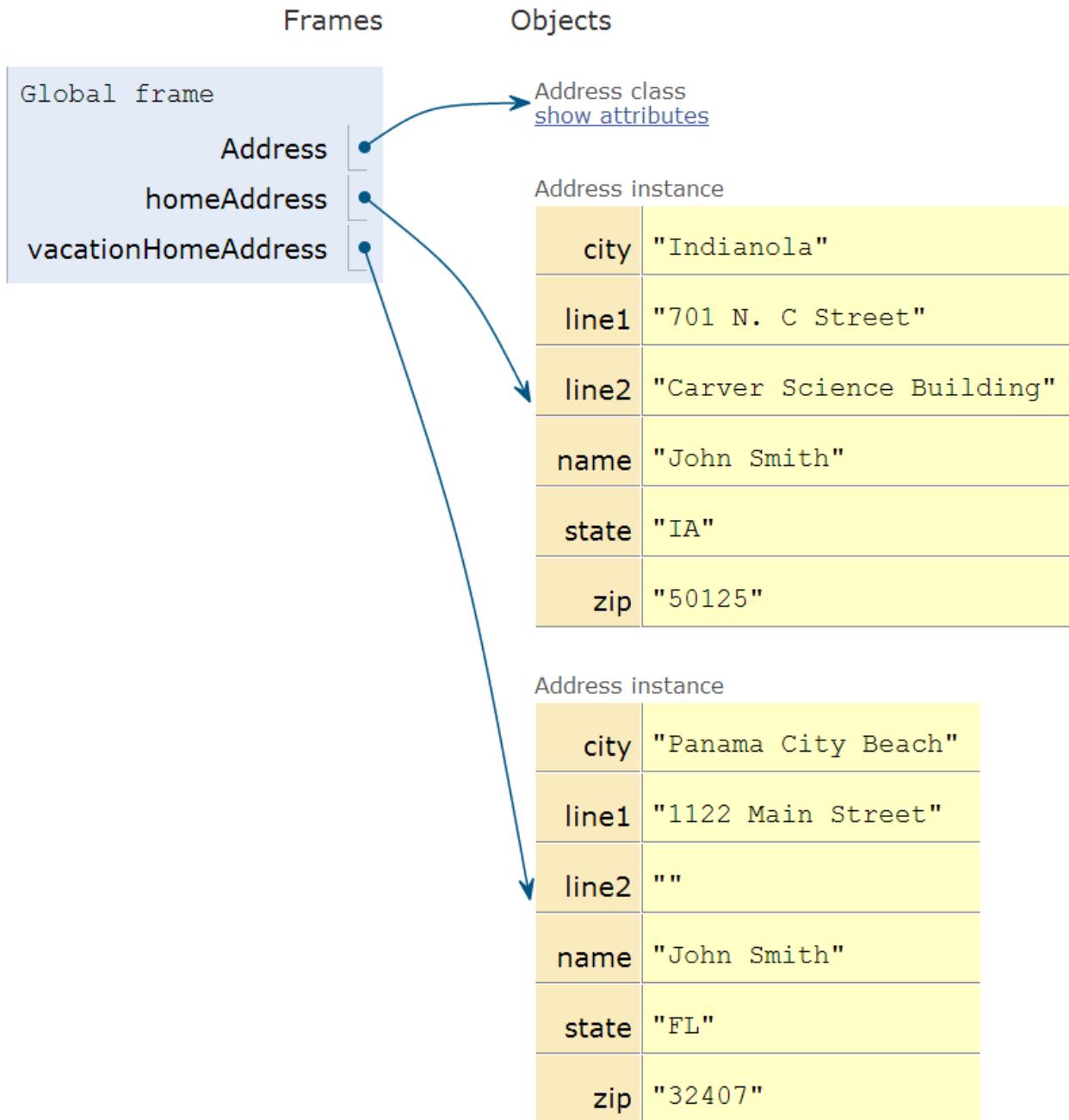


Fig. 14: Two Addresses

(continued from previous page)

```

12
13 def main():
14     print_address(home_address)
15     print()
16     print_address(vacation_home_address)
17
18
19 main()

```

1.14.3 Adding Methods to Classes

In addition to attributes, classes may have methods. A method is a function that exists inside of a class. Expanding the earlier example of a Dog class from the review problem 1 above, the code below adds a method for a dog barking.

```

1 class Dog():
2     def __init__(self):
3         self.age = 0
4         self.name = ""
5         self.weight = 0
6
7     def bark(self):
8         print("Woof")

```

The method definition is contained in lines 7-8 above. Method definitions in a class look almost exactly like function definitions. The big difference is the addition of a parameter `self` on line 7. The first parameter of any method in a class must be `self`. This parameter is required even if the function does not use it.

Here are the important items to keep in mind when creating methods for classes:

- Attributes should be listed first, methods after.
- The first parameter of any method must be `self`.
- Method definitions are indented exactly one tab stop.

Methods may be called in a manner similar to referencing attributes from an object. See the example code below.

```

1 my_dog = Dog()
2
3 my_dog.name = "Spot"
4 my_dog.weight = 20
5 my_dog.age = 3
6
7 my_dog.bark()

```

Line 1 creates the dog. Lines 3-5 set the attributes of the object. Line 7 calls the `bark` function. Note that even though the `bark` function has one parameter, `self`, the call does not pass in anything. This is because the first parameter is assumed to be a reference to the dog object itself. Behind the scenes, Python makes a call that looks like:

```

# Example, not actually legal
Dog.bark(my_dog)

```

If the `bark` function needs to make reference to any of the attributes, then it does so using the `self` reference variable. For example, we can change the `Dog` class so that when the dog barks, it also prints out the dog's name. In the code below, the `name` attribute is accessed using a dot operator and the `self` reference.

```
def bark(self):
    print("Woof says", self.name)
```

Attributes are adjectives, and methods are verbs. The drawing for the class would look like Figure 12.3.

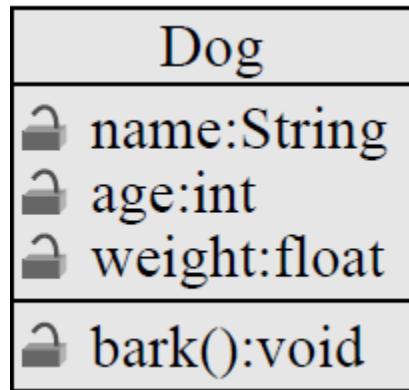


Fig. 15: Dog Class

Example: Ball Class

This example code could be used in Python/Arcade to draw a ball. Having all the parameters contained in a class makes data management easier. The diagram for the Ball class is shown in the figure below.

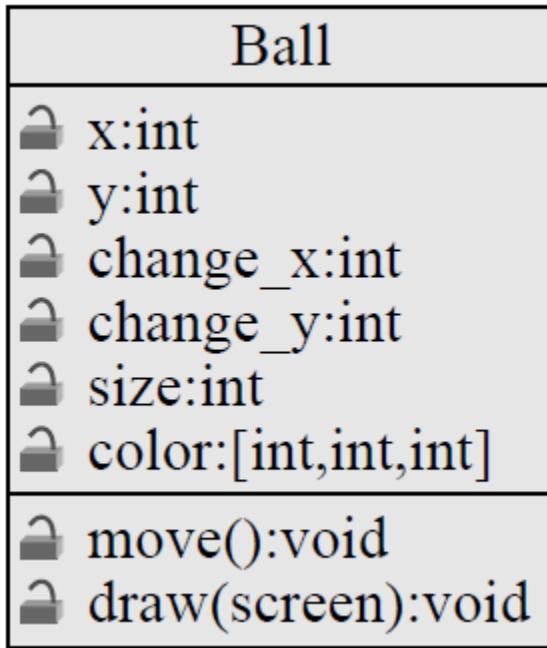


Fig. 16: Ball Class

```
1 class Ball():
2     def __init__(self):
3         # --- Class Attributes ---
```

(continues on next page)

(continued from previous page)

```

4     # Ball position
5     self.x = 0
6     self.y = 0
7
8     # Ball's vector
9     self.change_x = 0
10    self.change_y = 0
11
12    # Ball size
13    self.size = 10
14
15    # Ball color
16    self.color = [255, 255, 255]
17
18    # --- Class Methods ---
19    def move(self):
20        self.x += self.change_x
21        self.y += self.change_y
22
23    def draw(self):
24        arcade.draw_circle_filled(self.x, self.y, self.size, self.color )

```

Below is the code that would go ahead of the main program loop to create a ball and set its attributes:

```

1 the_ball = Ball()
2 the_ball.x = 100
3 the_ball.y = 100
4 the_ball.change_x = 2
5 the_ball.change_y = 1
6 the_ball.color = [255, 0, 0]

```

This code would go inside the main loop to move and draw the ball:

```

1 the_ball.move()
2 the_ball.draw()

```

1.14.4 References

Here's where we separate the true programmers from the want-to-be's. Understanding class references. Take a look at the following code:

```

1 class Person():
2     def __init__(self):
3         self.name = ""
4         self.money = 0
5
6
7     def main():
8         bob = Person()
9         bob.name = "Bob"
10        bob.money = 100
11
12        nancy = Person()
13        nancy.name = "Nancy"
14

```

(continues on next page)

(continued from previous page)

```

15 print(bob.name, "has", bob.money, "dollars.")
16 print(nancy.name, "has", nancy.money, "dollars.")
17
18
19 main()

```

The code above creates two instances of the Person () class, and using www.pythontutor.com we can visualize the two classes in the figure.

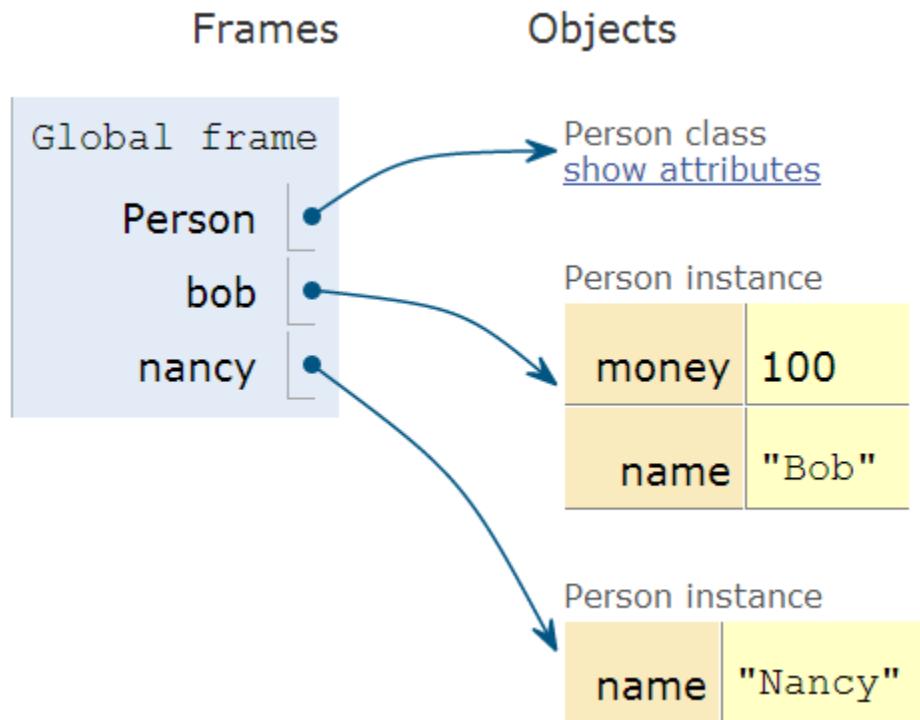


Fig. 17: Two Persons

The code above has nothing new. But the code below does:

```

1 class Person():
2     def __init__(self):
3         self.name = ""
4         self.money = 0
5
6
7 def main():
8     bob = Person()
9     bob.name = "Bob"
10    bob.money = 100
11
12    nancy = bob
13    nancy.name = "Nancy"
14
15    print(bob.name, "has", bob.money, "dollars.")
16    print(nancy.name, "has", nancy.money, "dollars.")
17

```

(continues on next page)

(continued from previous page)

```
18
19 main()
```

See the difference on line 10?

A common misconception when working with objects is to assume that the variable `bob` is the `Person` object. This is not the case. The variable `bob` is a *reference* to the `Person` object. That is, it stores the memory address of where the object is, and not the object itself.

If `bob` actually was the object, then line 9 could create a *copy* of the object and there would be two objects in existence. The output of the program would show both Bob and Nancy having 100 dollars. But when run, the program outputs the following instead:

```
Nancy has 100 dollars.
Nancy has 100 dollars.
```

What `bob` stores is a *reference* to the object. Besides reference, one may call this *address*, *pointer*, or *handle*. A reference is an address in computer memory for where the object is stored. This address is a hexadecimal number which, if printed out, might look something like `0x1e504`. When line 9 is run, the address is copied rather than the entire object the address points to. See the figure below.

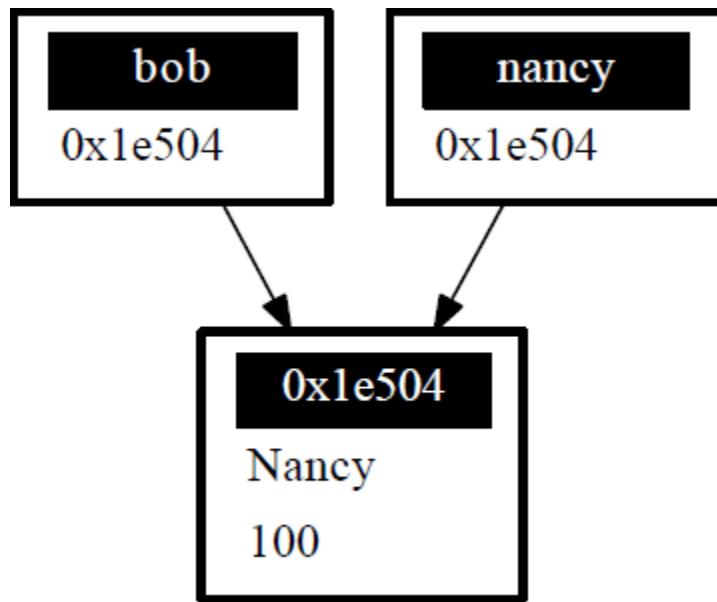


Fig. 18: Class References

We can also run this in www.pythontutor.com to see how both of the variables are pointing to the same object.

Functions and References

Look at the code example below. Line 1 creates a function that takes in a number as a parameter. The variable `money` is a variable that contains a copy of the data that was passed in. Adding 100 to that number does not change the number that was stored in `bob.money` on line 11. Thus, the print statement on line 14 prints out 100, and not 200.

```
1 def give_money1(money):
2     money += 100
3
```

(continues on next page)

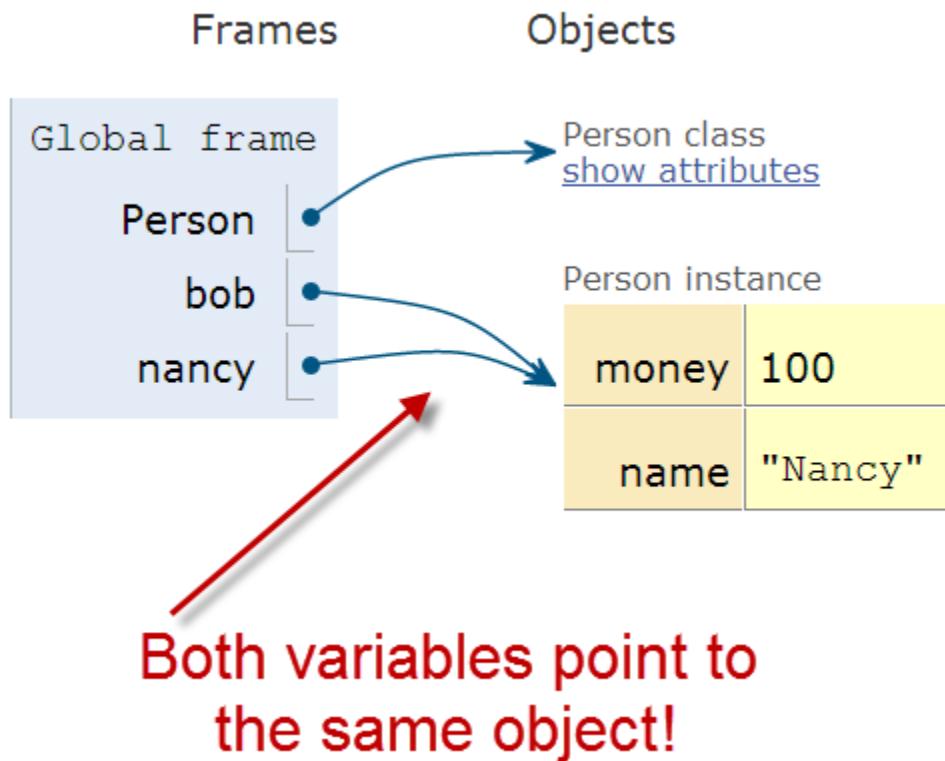


Fig. 19: One Person, Two Pointers

(continued from previous page)

```

4
5  class Person():
6      def __init__(self):
7          self.name = ""
8          self.money = 0
9
10
11 def main():
12     bob = Person()
13     bob.name = "Bob"
14     bob.money = 100
15
16     give_money1(bob.money)
17     print(bob.money)
18
19 main()

```

Running on PythonTutor we see that there are two instances of the `money` variable. One is a copy and local to the `give_money1` function.

Look at the additional code below. This code does cause `bob.money` to increase and the `print` statement to print 200.

```

1 def give_money2(person):
2     person.money += 100
3

```

(continues on next page)

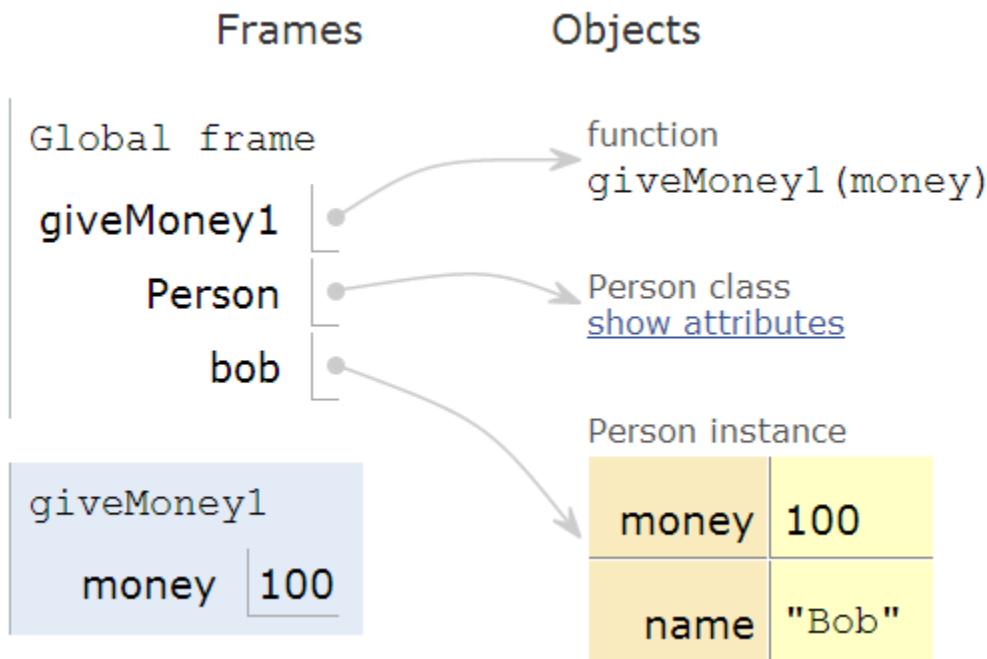


Fig. 20: Function References

(continued from previous page)

```

4 give_money2(bob)
5 print(bob.money)

```

Why is this? Because person contains a copy of the memory address of the object, not the actual object itself. One can think of it as a bank account number. The function has a copy of the bank account number, not a copy of the whole bank account. So using the copy of the bank account number to deposit 100 dollars causes Bob's bank account balance to go up.

Arrays work the same way. A function that takes in an array (list) as a parameter and modifies values in that array will be modifying the same array that the calling code created. The address of the array is copied, not the entire array.

Review Questions

1. Create a class called Cat. Give it attributes for name, color, and weight. Give it a method called meow.
2. Create an instance of the cat class, set the attributes, and call the meow method.
3. Create a class called Monster. Give it an attribute for name and an integer attribute for health. Create a method called decrease_health that takes in a parameter amount and decreases the health by that much. Inside that method, print that the animal died if health goes below zero.

1.14.5 Constructors

There's a terrible problem with our class for Dog listed below. When we create a dog, by default the dog has no name. Dogs should have names! We should not allow dogs to be born and then never be given a name. Yet the code below allows this to happen, and that dog will never have a name.

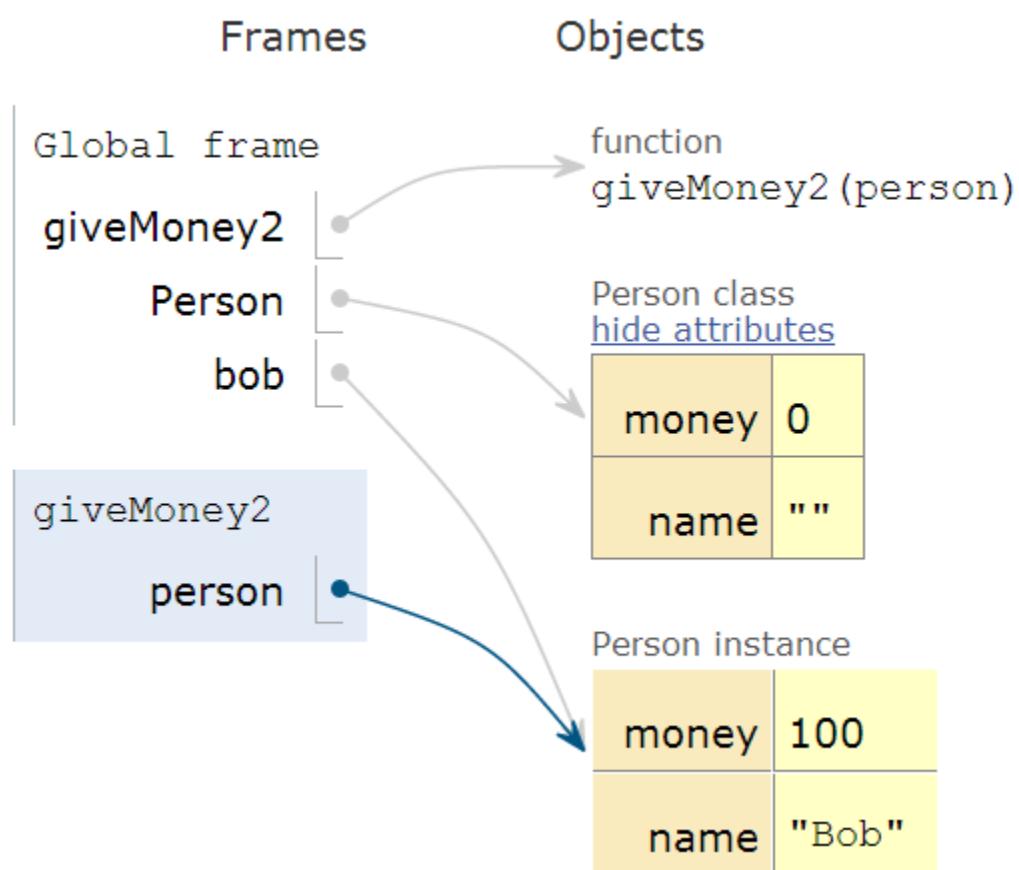


Fig. 21: Function References

```
1 class Dog():
2     def __init__(self):
3         self.name = ""
4
5
6 def main():
7     my_dog = Dog()
8
9
10 main()
```

Python doesn't want this to happen. That's why Python classes have a special function that is called any time an instance of that class is created. By adding a function called a constructor, a programmer can add code that is automatically run each time an instance of the class is created. See the example constructor code below:

Listing 53: Example of a class with a constructor

```
1 class Dog():
2     def __init__(self):
3         """ Constructor. Called when creating an object of this type. """
4         self.name = ""
5         print("A new dog is born!")
6
7
8 def main():
9     # This creates the dog
10    my_dog = Dog()
```

The constructor starts on line 2. It must be named `__init__`. There are two underscores before the init, and two underscores after. A common mistake is to only use one.

The constructor must take in `self` as the first parameter just like other methods in a class. When the program is run, it will print:

```
.. code-block:: text
```

A new dog is born!

When a `Dog` object is created on line 8, the `__init__` function is automatically called and the message is printed to the screen.

Avoid This Mistake

Put everything for a method into just one definition. Don't define it twice. For example:

```
1 # Wrong:
2 class Dog():
3     def __init__(self):
4         self.age = 0
5         self.name = ""
6         self.weight = 0
7
8     def __init__(self):
9         print("New dog!")
```

The computer will just ignore the first `__init__` and go with the last definition. Instead do this:

```

1 # Correct:
2 class Dog():
3     def __init__(self):
4         self.age = 0
5         self.name = ""
6         self.weight = 0
7         print("New dog!")

```

A constructor can be used for initializing and setting data for the object. The example Dog class above still allows the name attribute to be left blank after the creation of the dog object. How do we keep this from happening? Many objects need to have values right when they are created. The constructor function can be used to make this happen. See the code below:

Listing 54: Constructor that takes in data to initialize the class

```

1 class Dog():
2
3     def __init__(self, new_name):
4         """ Constructor. """
5         self.name = new_name
6
7
8     def main():
9         # This creates the dog
10        my_dog = Dog("Spot")
11
12        # Print the name to verify it was set
13        print(my_dog.name)
14
15        # This line will give an error because
16        # a name is not passed in.
17        her_dog = Dog()
18
19 main()

```

On line 3 the constructor function now has an additional parameter named `new_name`. The value of this parameter is used to set the name attribute in the `Dog` class on line 8. It is no longer possible to create a `Dog` class without a name. The code on line 15 tries this. It will cause a Python error and it will not run. A common mistake is to name the parameter of the `__init__` function the same as the attribute and assume that the values will automatically synchronize. This does not happen.

Review Questions

- Should class names begin with an upper or lower case letter?
- Should method names begin with an upper or lower case letter?
- Should attribute names begin with an upper or lower case letter?
- Which should be listed first in a class, attributes or methods?
- What are other names for a reference?
- What is another name for instance variable?
- What is the name for an instance of a class?
- Create a class called Star that will print out “A star is born!” every time it is created.

- Create a class called Monster with attributes for health and a name. Add a constructor to the class that sets the health and name of the object with data passed in as parameters.

1.14.6 Inheritance

Another powerful feature of using classes and objects is the ability to make use of *inheritance*. It is possible to create a class and inherit all of the attributes and methods of a *parent class*.

For example, a program may create a class called Boat which has all the attributes needed to represent a boat in a game:

Listing 55: Class definition for a boat

```
1 class Boat():
2     def __init__(self):
3         self.tonnage = 0
4         self.name = ""
5         self.is_docked = True
6
7     def dock(self):
8         if self.is_docked:
9             print("You are already docked.")
10        else:
11            self.is_docked = True
12            print("Docking")
13
14    def undock(self):
15        if not self.is_docked:
16            print("You aren't docked.")
17        else:
18            self.is_docked = False
19            print("Undocking")
```

To test out our code:

Listing 56: Floating our boat

```
1 b = Boat()
2
3 b.dock()
4 b.undock()
5 b.undock()
6 b.dock()
7 b.dock()
```

The output:

```
You are already docked.
Undocking
You aren't docked.
Docking
You are already docked.
```

(If you watch the video for this section of the class, you'll note that the "Boat" class in the video doesn't actually run. The code above has been corrected but I haven't fixed the video. Use this as a reminder, no matter how simple the code and how experienced the developer, test your code before you deliver it!)

Our program also needs a submarine. Our submarine can do everything a boat can, plus we need a command for submerge. Without inheritance we have two options.

- One, add the `submerge()` command to our boat. This isn't a great idea because we don't want to give the impression that our boats normally submerge.
- Two, we could create a copy of the `Boat` class and call it `Submarine`. In this class we'd add the `submerge()` command. This is easy at first, but things become harder if we change the `Boat` class. A programmer would need to remember that we'd need to change not only the `Boat` class, but also make the same changes to the `Submarine` class. Keeping this code synchronized is time consuming and error-prone.

Luckily, there is a better way. Our program can create *child classes* that will inherit all the attributes and methods of the *parent class*. The child classes may then add fields and methods that correspond to their needs. For example:

```
1 class Submarine(Boat):
2     def submerge(self):
3         print("Submerge!")
```

Line 1 is the important part. Just by putting `Boat` in between the parentheses during the class declaration, we have automatically picked up every attribute and method that is in the `Boat` class. If we update `Boat`, then the child class `Submarine` will automatically get these updates. Inheritance is that easy!

The next code example is diagrammed out in the figure below.

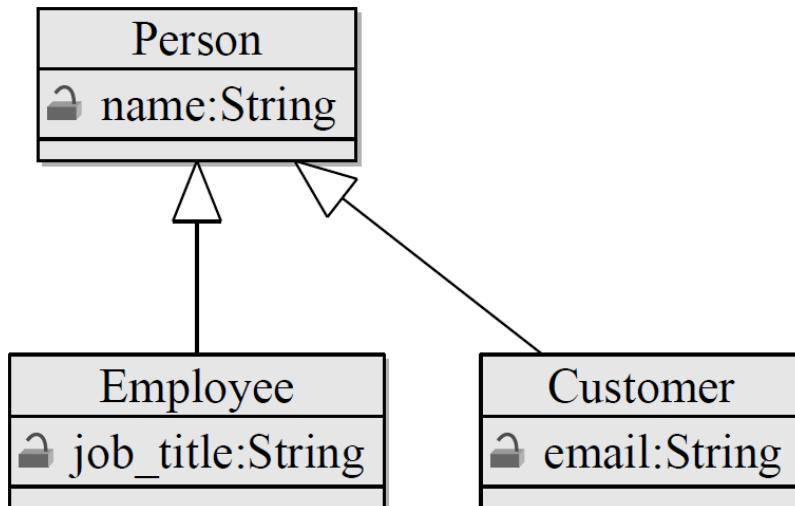


Fig. 22: Class Diagram

Listing 57: Person, Employee, Customer Classes Examples

```
1 class Person():
2     def __init__(self):
3         self.name = ""
4
5 class Employee(Person):
6     def __init__(self):
7         # Call the parent/super class constructor first
8         super().__init__()
9
10        # Now set up our variables
11        self.job_title = ""
```

(continues on next page)

(continued from previous page)

```

13 class Customer(Person):
14     def __init__(self):
15         super().__init__()
16         self.email = ""
17
18     def main():
19         john_smith = Person()
20         john_smith.name = "John Smith"
21
22         jane_employee = Employee()
23         jane_employee.name = "Jane Employee"
24         jane_employee.job_title = "Web Developer"
25
26         bob_customer = Customer()
27         bob_customer.name = "Bob Customer"
28         bob_customer.email = "send_me@spam.com"
29
30     main()

```

By placing `Person` between the parentheses on lines 5 and 13, the programmer has told the computer that `Person` is a parent class to both `Employee` and `Customer`. This allows the program to set the name attribute on lines 19 and 22.

Methods are also inherited. Any method the parent has, the child class will have too. But what if we have a method in both the child and parent class?

We have two options. We can run them both with `super()` keyword. Using `super()` followed by a dot operator, and then finally a method name allows you to call the parent's version of the method.

The code above shows the first option using `super` where we run not only the child constructor but also the parent constructor.

If you are writing a method for a child and want to call a parent method, normally it will be the first statement in the child method. Notice how it is in the example above.

All constructors should call the parent constructor because then you'd have a child without a parent and that is just sad. In fact, some languages force this rule, but Python doesn't.

The second option? Methods may be overridden by a child class to provide different functionality. The example below shows both options. The `Employee.report` overrides the `Person.report` because it never calls and runs the parent `report` method. The `Customer.report` does call the parent and the `report` method in `Customer` adds to the `Person` functionality.

Listing 58: Overriding constructors

```

1 class Person():
2     def __init__(self):
3         self.name = ""
4
5     def report(self):
6         # Basic report
7         print("Report for", self.name)
8
9 class Employee(Person):
10    def __init__(self):
11        # Call the parent/super class constructor first
12        super().__init__()
13

```

(continues on next page)

(continued from previous page)

```

14     # Now set up our variables
15     self.job_title = ""
16
17     def report(self):
18         # Here we override report and just do this:
19         print("Employee report for", self.name)
20
21 class Customer(Person):
22     def __init__(self):
23         super().__init__()
24         self.email = ""
25
26     def report(self):
27         # Run the parent report:
28         super().report()
29         # Now add our own stuff to the end so we do both
30         print("Customer e-mail:", self.email)
31
32 def main():
33     john_smith = Person()
34     john_smith.name = "John Smith"
35
36     jane_employee = Employee()
37     jane_employee.name = "Jane Employee"
38     jane_employee.job_title = "Web Developer"
39
40     bob_customer = Customer()
41     bob_customer.name = "Bob Customer"
42     bob_customer.email = "send_me@spam.com"
43
44     john_smith.report()
45     jane_employee.report()
46     bob_customer.report()
47
48 main()

```

Is-A and Has-A Relationships

Classes have two main types of relationships. They are “is a” and “has a” relationships.

A parent class should always be a more general, abstract version of the child class. This type of child to parent relationship is called an *is a* relationship. For example, a parent class `Animal` could have a child class `Dog`. The dog *is an* animal. The `Dog` class could have a child class `Poodle`. The poodle *is a* dog, and *is an* animal.

It does not work the other way! A dolphin *is a* mammal, but a mammal is not always a dolphin. So the class `Dolphin` should never be a parent to a class `Mammal`.

Unrelated items that do not pass the *is a* test should not form parent/child relationships. For example, a class `Table` should not be a parent to a class `Chair` because a chair is not a table.

The other type of relationship is the *has a* relationship. These relationships are implemented in code by class attributes. A dog has a name, and so the `Dog` class has an attribute for name. Likewise a person could have a dog, and that would be implemented by having the `Person` class have an attribute for `Dog`. The `Person` class would not derive from `Dog` because that would be some kind of insult.

Looking at the prior code example we can see:

- Employee is a person.
- Customer is a person.
- Person has a name.
- Employee has a job title.
- Customer has an e-mail.

1.14.7 Static Variables vs. Instance Variables

The difference between static and instance variables is confusing. Thankfully it isn't necessary to completely understand the difference right now. But if you stick with programming, it will be. Therefore we will briefly introduce it here.

There are also some oddities with Python that kept me confused the first several years I've made this book available. So you might see older videos and examples where I get it wrong.

An *instance variable* is the type of class variable we've used so far. Each instance of the class gets its own value. For example, in a room full of people each person will have their own age. Some of the ages may be the same, but we still need to track each age individually.

With instance variables, we can't just say "age" with a room full of people. We need to specify *whose* age we are talking about. Also, if there are no people in the room, then referring to an age when there are no people to have an age makes no sense.

With *static variables* the value is the same for every single instance of the class. Even if there are no instances, there still is a value for a static variable. For example, we might have a `count` static variable for the number of Human classes in existence. No humans? The value is zero, but the count variable still exists.

In the example below, `ClassA` creates an instance variable. `ClassB` creates a static variable.

```
1 # Example of an instance variable
2 class ClassA():
3     def __init__(self):
4         self.y = 3
5
6 # Example of a static variable
7 class ClassB():
8     x = 7
9
10 def main():
11     # Create class instances
12     a = ClassA()
13     b = ClassB()
14
15     # Two ways to print the static variable.
16     # The second way is the proper way to do it.
17     print(b.x)
18     print(ClassB.x)
19
20     # One way to print an instance variable.
21     # The second generates an error, because we don't know what instance
22     # to reference.
23     print(a.y)
24     print(ClassA.y)
25
26 main()
```

In the example above, lines 16 and 17 print out the static variable. Line 17 is the “proper” way to do so. Unlike before, we can refer to the class name when using static variables, rather than a variable that points to a particular instance. Because we are working with the class name, by looking at line 17 we instantly can tell we are working with a static variable. Line 16 could be either an instance or static variable. That confusion makes line 17 the better choice.

Line 22 prints out the instance variable, just like we’ve done in prior examples. Line 23 will generate an error because each instance of `y` is different (it is an instance variable after all) and we aren’t telling the computer what instance of `ClassA` we are talking about.

Instance Variables Hiding Static Variables

This is one “feature” of Python I dislike. It is possible to have a static variable, and an instance variable *with the same name*. Look at the example below:

```

1 # Class with a static variable
2 class ClassB():
3     x = 7
4
5 def main():
6     # Create a class instance
7     b = ClassB()
8
9     # This prints 7
10    print(b.x)
11
12    # This also prints 7
13    print(ClassB.x)
14
15    # Set x to a new value using the class name
16    ClassB.x = 8
17
18    # This also prints 8
19    print(b.x)
20
21    # This prints 8
22    print(ClassB.x)
23
24    # Set x to a new value using the instance.
25    # Wait! Actually, it doesn't set x to a new value!
26    # It creates a brand new variable, x. This x
27    # is an instance variable. The static variable is
28    # also called x. But they are two different
29    # variables. This is super-confusing and is bad
30    # practice.
31    b.x = 9
32
33    # This prints 9
34    print(b.x)
35
36    # This prints 8. NOT 9!!!
37    print(ClassB.x)
38
39 main()
```

Allowing instance variables to hide static variable caused confusion for me for many years!

1.15 Using the Window Class

We can use a class to represent our program. The Arcade library has a built-in class that represents a window on the screen. We can create our own child class and override functions to handle:

- Start-up and initialization
- Drawing the items on our screen
- Animating/Updating the positions of items on our screen
- Responding to the keyboard
- Responding to the mouse

One of the best ways of learning to program is to look at sample code. This chapter has several examples designed to learn how to:

- Open a window using an object-oriented approach
- Animating objects
- Moving objects with the mouse
- Moving objects with the keyboard
- Moving objects with the joystick

1.15.1 Creating a Window with a Class

Up to now, we have used a function called `open_window` to open a window. Here's the code:

Listing 59: `open_window_with_function.py`

```
1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5
6
7 def main():
8     arcade.open_window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing Example")
9
10    arcade.run()
11
12
13 main()
```

We can also create an instance of a class called `Window` to open a window. The code is rather straight-forward:

Listing 60: `open_window_with_object.py`

```
1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5
6
7 def main():
8     window = arcade.Window(SCREEN_WIDTH, SCREEN_HEIGHT, "Drawing Example")
```

(continues on next page)

(continued from previous page)

```

9     arcade.run()
10
11
12
13 main()
```

Function calls, and calls to create an instance of an object look very similar. The tell-tale clue that we are creating an instance of an object in the second example is the fact that `Window` is capitalized.

1.15.2 Extending the Window Class

Arcade's `Window` class has a lot of built-in methods that are automatically called when needed. Methods for drawing, for responding to the keyboard, the mouse, and more. You can see all the methods by looking at the [Window Class Documentation](#). But by default, these methods don't do anything. We need to change that.

As we learned from the prior chapter, we can extend the functionality of a class by creating a child class. Therefore, we can extend the `Window` class by creating a child class of it. I'm going to call my child class `MyGame`.

Listing 61: extending_window_class.py

```

1 import arcade
2
3
4 class MyGame(arcade.Window):
5
6     def __init__(self, width, height, title):
7
8         # Call the parent class's init function
9         super().__init__(width, height, title)
10
11
12 def main():
13     window = MyGame(640, 480, "Drawing Example")
14
15     arcade.run()
16
17
18 main()
```

1.15.3 Drawing with the Window Class

To draw with the `Window` class, we need to create our own method called `on_draw`. This will override the default `on_draw` method built into the `Window` class. We will put our drawing code in there.

The `on_draw` method gets called about 60 times per second. We'll use this fact when we do animation.

We also need to set the background color. Since we only need to do this once, we will do that in the `__init__` method. No sense setting the background 60 times per second when it isn't changing.

Listing 62: drawing.py

```

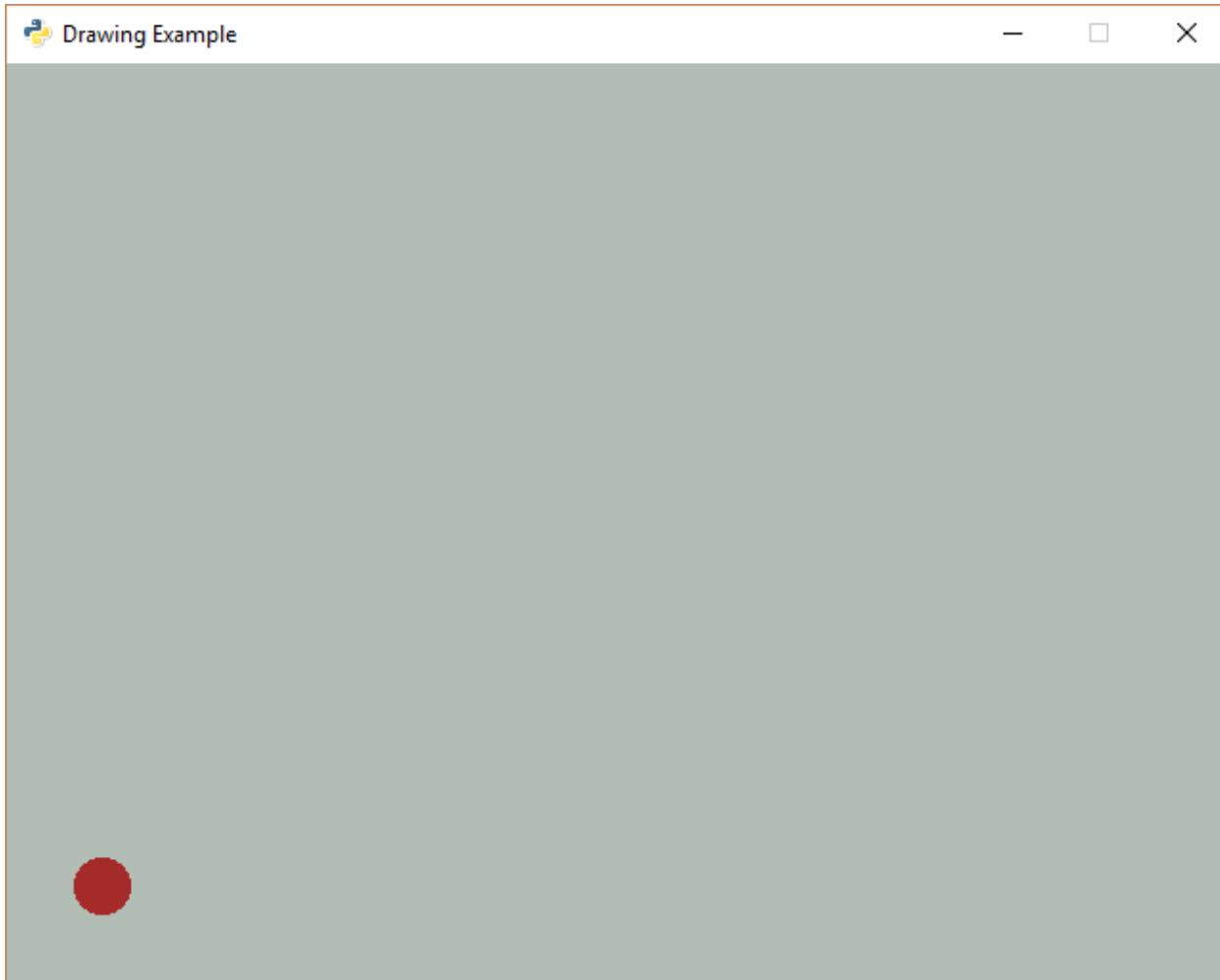
1 import arcade
2
3
```

(continues on next page)

(continued from previous page)

```
4 class MyGame(arcade.Window):
5
6     def __init__(self, width, height, title):
7
8         # Call the parent class's init function
9         super().__init__(width, height, title)
10
11        # Set the background color
12        arcade.set_background_color(arcade.color.ASH_GREY)
13
14    def on_draw(self):
15        """ Called whenever we need to draw the window. """
16        arcade.start_render()
17
18        arcade.draw_circle_filled(50, 50, 15, arcade.color.AUBURN)
19
20
21    def main():
22        window = MyGame(640, 480, "Drawing Example")
23
24        arcade.run()
25
26
27 main()
```

The result of this program just looks like:



1.15.4 Animating

By overriding the `update` method, we can update our ball position and animate our scene:

Listing 63: simple_animation.py

```
1 import arcade
2
3
4 class MyGame(arcade.Window):
5
6     def __init__(self, width, height, title):
7
8         # Call the parent class's init function
9         super().__init__(width, height, title)
10
11        # Set the background color
12        arcade.set_background_color(arcade.color.ASH_GREY)
13
14        # Attributes to store where our ball is
15        self.ball_x = 50
16        self.ball_y = 50
```

(continues on next page)

(continued from previous page)

```

17
18     def on_draw(self):
19         """ Called whenever we need to draw the window. """
20         arcade.start_render()
21
22         arcade.draw_circle_filled(self.ball_x, self.ball_y, 15, arcade.color.AUBURN)
23
24     def update(self, delta_time):
25         """ Called to update our objects. Happens approximately 60 times per second."""
26         ←
27         self.ball_x += 1
28         self.ball_y += 1
29
30 def main():
31     window = MyGame(640, 480, "Drawing Example")
32
33     arcade.run()
34
35
36 main()

```

Encapsulating Our Animation Object

It doesn't take much imagination to realize that adding more parameters to the ball, getting it to bounce, or even having several balls on the screen would make our `MyApplication` class very complex.

If only there was a way to encapsulate all that “ball” stuff together. Wait! There is! Using classes!

Here is a more complex example, but all the logic for the ball has been moved into a new `Ball` class.

Listing 64: `ball_class_example.py`

```

1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5
6
7 class Ball:
8     """ This class manages a ball bouncing on the screen. """
9
10    def __init__(self, position_x, position_y, change_x, change_y, radius, color):
11        """ Constructor. """
12
13        # Take the parameters of the init function above, and create instance
14        ←variables out of them.
15        self.position_x = position_x
16        self.position_y = position_y
17        self.change_x = change_x
18        self.change_y = change_y
19        self.radius = radius
20        self.color = color
21
22    def draw(self):
23        """ Draw the balls with the instance variables we have. """

```

(continues on next page)

(continued from previous page)

```

23     arcade.draw_circle_filled(self.position_x, self.position_y, self.radius, self.
24     ↪color)
25
26     def update(self):
27         """ Code to control the ball's movement. """
28
29         # Move the ball
30         self.position_y += self.change_y
31         self.position_x += self.change_x
32
33         # See if the ball hit the edge of the screen. If so, change direction
34         if self.position_x < self.radius:
35             self.change_x *= -1
36
37         if self.position_x > SCREEN_WIDTH - self.radius:
38             self.change_x *= -1
39
40         if self.position_y < self.radius:
41             self.change_y *= -1
42
43         if self.position_y > SCREEN_HEIGHT - self.radius:
44             self.change_y *= -1
45
46 class MyGame(arcade.Window):
47     """ My window class. """
48
49     def __init__(self, width, height, title):
50         """ Constructor. """
51
52         # Call the parent class's init function
53         super().__init__(width, height, title)
54         arcade.set_background_color(arcade.color.ASH_GREY)
55
56         # Create our ball
57         self.ball = Ball(50, 50, 3, 3, 15, arcade.color.AUBURN)
58
59     def on_draw(self):
60         """ Called whenever we need to draw the window. """
61         arcade.start_render()
62         self.ball.draw()
63
64     def update(self, delta_time):
65         """ Called to update our objects. Happens approximately 60 times per second. """
66         ↪
67         self.ball.update()
68
69     def main():
70         window = MyGame(640, 480, "Drawing Example")
71
72         arcade.run()
73
74
75 main()

```

Here it is in action:

Animating a List

Wouldn't it be nice to animate multiple items? How do we track multiple items? With a list! This takes our previous example and animates three balls at once.

Listing 65: ball_list_example.py

```
1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5
6
7 class Ball:
8     """ This class manages a ball bouncing on the screen. """
9
10    def __init__(self, position_x, position_y, change_x, change_y, radius, color):
11        """ Constructor. """
12
13        # Take the parameters of the init function above, and create instance_
14        # variables out of them.
15        self.position_x = position_x
16        self.position_y = position_y
17        self.change_x = change_x
18        self.change_y = change_y
19        self.radius = radius
20        self.color = color
21
22    def draw(self):
23        """ Draw the balls with the instance variables we have. """
24        arcade.draw_circle_filled(self.position_x, self.position_y, self.radius, self.
25        color)
26
27    def update(self):
28        """ Code to control the ball's movement. """
29
30        # Move the ball
31        self.position_y += self.change_y
32        self.position_x += self.change_x
33
34        # See if the ball hit the edge of the screen. If so, change direction
35        if self.position_x < self.radius:
36            self.change_x *= -1
37
38        if self.position_x > SCREEN_WIDTH - self.radius:
39            self.change_x *= -1
40
41        if self.position_y < self.radius:
42            self.change_y *= -1
43
44        if self.position_y > SCREEN_HEIGHT - self.radius:
45            self.change_y *= -1
46
47 class MyGame(arcade.Window):
```

(continues on next page)

(continued from previous page)

```

47
48     def __init__(self, width, height, title):
49
50         # Call the parent class's init function
51         super().__init__(width, height, title)
52         arcade.set_background_color(arcade.color.ASH_GREY)
53
54         # Create a list for the balls
55         self.ball_list = []
56
57         # Add three balls to the list
58         ball = Ball(50, 50, 3, 3, 15, arcade.color.AUBURN)
59         self.ball_list.append(ball)
60
61         ball = Ball(100, 150, 2, 3, 15, arcade.color.PURPLE_MOUNTAIN_MAESTY)
62         self.ball_list.append(ball)
63
64         ball = Ball(150, 250, -3, -1, 15, arcade.color.FOREST_GREEN)
65         self.ball_list.append(ball)
66
67     def on_draw(self):
68         """ Called whenever we need to draw the window. """
69         arcade.start_render()
70
71         # Use a "for" loop to pull each ball from the list, then call the draw
72         # method on that ball.
73         for ball in self.ball_list:
74             ball.draw()
75
76     def update(self, delta_time):
77         """ Called to update our objects. Happens approximately 60 times per second. """
78
79         # Use a "for" loop to pull each ball from the list, then call the update
80         # method on that ball.
81         for ball in self.ball_list:
82             ball.update()
83
84
85     def main():
86         window = MyGame(640, 480, "Drawing Example")
87
88         arcade.run()
89
90
91 main()

```

1.16 User Control

How do we interact with the user? Get the user to move an object on the screen?

We can do this with the mouse, with the keyboard, or with the game controller.

1.16.1 Move with the Mouse

The key to managing mouse motion to override the `on_mouse_motion` in the `arcade.Window` class. That method is called every time the mouse moves. The method definition looks like this:

```
def on_mouse_motion(self, x, y, dx, dy):
```

The `x` and `y` are the coordinates of the mouse. the `dx` and `dy` represent the change in `x` and `y` since the last time the method was called.

Often when controlling a graphical item on the screen with the mouse, we do not want to see the mouse pointer. If you don't want to see the mouse pointer, in the `__init__` method, call the following method in the parent class:

```
self.set_mouse_visible(False)
```

The example below takes our `Ball` class, and moves it around the screen with the mouse.

Listing 66: move_with_mouse.py

```
1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5
6
7 class Ball:
8     def __init__(self, position_x, position_y, radius, color):
9
10         # Take the parameters of the init function above, and create instance_
11         #variables out of them.
12         self.position_x = position_x
13         self.position_y = position_y
14         self.radius = radius
15         self.color = color
16
17     def draw(self):
18         """ Draw the balls with the instance variables we have. """
19         arcade.draw_circle_filled(self.position_x, self.position_y, self.radius,
20         self.color)
21
22
23 class MyGame(arcade.Window):
24
25     def __init__(self, width, height, title):
26
27         # Call the parent class's init function
28         super().__init__(width, height, title)
29
30         # Make the mouse disappear when it is over the window.
31         # So we just see our object, not the pointer.
32         self.set_mouse_visible(False)
33
34         arcade.set_background_color(arcade.color.ASH_GREY)
35
36         # Create our ball
37         self.ball = Ball(50, 50, 15, arcade.color.AUBURN)
38
39     def on_draw(self):
```

(continues on next page)

(continued from previous page)

```

38     """ Called whenever we need to draw the window. """
39     arcade.start_render()
40     self.ball.draw()
41
42     def on_mouse_motion(self, x, y, dx, dy):
43         """ Called to update our objects. Happens approximately 60 times per second."""
44         self.ball.position_x = x
45         self.ball.position_y = y
46
47
48     def main():
49         window = MyGame(640, 480, "Drawing Example")
50         arcade.run()
51
52
53     main()

```

1.16.2 Mouse Clicks

You can also process mouse clicks by defining an `on_mouse_press` method:

```

def on_mouse_press(self, x, y, button, modifiers):
    """ Called when the user presses a mouse button. """

    if button == arcade.MOUSE_BUTTON_LEFT:
        print("Left mouse button pressed at", x, y)
    elif button == arcade.MOUSE_BUTTON_RIGHT:
        print("Right mouse button pressed at", x, y)

```

1.16.3 Move with the Keyboard

Moving with the game controller is similar to our bouncing ball example. There are just two differences:

- We control the `change_x` and `change_y` with the keyboard
- When we hit the edge of the screen we stop, rather than bounce.

To detect when a key is hit, we override the `on_key_press` method. We might think of hitting a key as one event. But it is actually two. When the key is pressed, we start moving. When the key is released we stop moving. That makes for two events. Releasing a key is controlled by `on_key_release`.

These methods have a `key` variable as a parameter that can be compared with an `if` statement to the values in the `arcade.key` library.

```

def on_key_press(self, key, modifiers):
    if key == arcade.key.LEFT:
        print("Left key hit")
    elif key == arcade.key.A:
        print("The 'a' key was hit")

```

We can use this in a program to move a ball around the screen. See the highlighted lines in the program below:

Listing 67: move_with_keyboard_simple.py

```
1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5 MOVEMENT_SPEED = 3
6
7
8 class Ball:
9     def __init__(self, position_x, position_y, change_x, change_y, radius, color):
10
11         # Take the parameters of the init function above, and create instance_
12         #variables out of them.
13         self.position_x = position_x
14         self.position_y = position_y
15         self.change_x = change_x
16         self.change_y = change_y
17         self.radius = radius
18         self.color = color
19
20     def draw(self):
21         """ Draw the balls with the instance variables we have. """
22         arcade.draw_circle_filled(self.position_x, self.position_y, self.radius, self.
23         color)
24
25     def update(self):
26         # Move the ball
27         self.position_y += self.change_y
28         self.position_x += self.change_x
29
30
31 class MyGame(arcade.Window):
32
33     def __init__(self, width, height, title):
34
35         # Call the parent class's init function
36         super().__init__(width, height, title)
37
38         # Make the mouse disappear when it is over the window.
39         # So we just see our object, not the pointer.
40         self.set_mouse_visible(False)
41
42         arcade.set_background_color(arcade.color.ASH_GREY)
43
44         # Create our ball
45         self.ball = Ball(50, 50, 0, 0, 15, arcade.color.AUBURN)
46
47     def on_draw(self):
48         """ Called whenever we need to draw the window. """
49         arcade.start_render()
50         self.ball.draw()
51
52     def update(self, delta_time):
53         self.ball.update()
54
55     def on_key_press(self, key, modifiers):
```

(continues on next page)

(continued from previous page)

```

54     """ Called whenever the user presses a key. """
55     if key == arcade.key.LEFT:
56         self.ball.change_x = -MOVEMENT_SPEED
57     elif key == arcade.key.RIGHT:
58         self.ball.change_x = MOVEMENT_SPEED
59     elif key == arcade.key.UP:
60         self.ball.change_y = MOVEMENT_SPEED
61     elif key == arcade.key.DOWN:
62         self.ball.change_y = -MOVEMENT_SPEED
63
64     def on_key_release(self, key, modifiers):
65         """ Called whenever a user releases a key. """
66         if key == arcade.key.LEFT or key == arcade.key.RIGHT:
67             self.ball.change_x = 0
68         elif key == arcade.key.UP or key == arcade.key.DOWN:
69             self.ball.change_y = 0
70
71
72 def main():
73     window = MyGame(640, 480, "Drawing Example")
74     arcade.run()
75
76
77 main()

```

Keep From Moving Off Screen

Unfortunately in the prior program, there is nothing that keeps the player from moving off-screen. If we want to stop the player from moving off-screen we need some additional code.

We detect the edge by comparing `position_x` with the left and right side of the screen For example:

```
if self.position_x < 0:
```

But this isn't perfect. Because the position specifies the *center* of the ball, by the time the x coordinate is 0 we are already have off the screen. It is better to compare it to the ball's radius:

```
if self.position_x < self.radius:
```

What do we do once it hits the edge? Just set the value back to the edge:

```
# See if the ball hit the edge of the screen. If so, change direction
if self.position_x < self.radius:
    self.position_x = self.radius
```

Here's a full example:

Listing 68: move_with_keyboard_edge_detect.py

```

1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5 MOVEMENT_SPEED = 3
6
```

(continues on next page)

(continued from previous page)

```

7   class Ball:
8       def __init__(self, position_x, position_y, change_x, change_y, radius, color):
9           # Take the parameters of the init function above, and create instance_
10          ↪variables out of them.
11          self.position_x = position_x
12          self.position_y = position_y
13          self.change_x = change_x
14          self.change_y = change_y
15          self.radius = radius
16          self.color = color
17
18      def draw(self):
19          """ Draw the balls with the instance variables we have. """
20          arcade.draw_circle_filled(self.position_x, self.position_y, self.radius, self.
21          ↪color)
22
23      def update(self):
24          # Move the ball
25          self.position_y += self.change_y
26          self.position_x += self.change_x
27
28          # See if the ball hit the edge of the screen. If so, change direction
29          if self.position_x < self.radius:
30              self.position_x = self.radius
31
32          if self.position_x > SCREEN_WIDTH - self.radius:
33              self.position_x = SCREEN_WIDTH - self.radius
34
35          if self.position_y < self.radius:
36              self.position_y = self.radius
37
38          if self.position_y > SCREEN_HEIGHT - self.radius:
39              self.position_y = SCREEN_HEIGHT - self.radius
40
41
42  class MyGame(arcade.Window):
43
44      def __init__(self, width, height, title):
45
46          # Call the parent class's init function
47          super().__init__(width, height, title)
48
49          # Make the mouse disappear when it is over the window.
50          # So we just see our object, not the pointer.
51          self.set_mouse_visible(False)
52
53          arcade.set_background_color(arcade.color.ASH_GREY)
54
55          # Create our ball
56          self.ball = Ball(50, 50, 0, 0, 15, arcade.color.AUBURN)
57
58      def on_draw(self):
59          """ Called whenever we need to draw the window. """
60          arcade.start_render()
61          self.ball.draw()

```

(continues on next page)

(continued from previous page)

```

62
63     def update(self, delta_time):
64         self.ball.update()
65
66     def on_key_press(self, key, modifiers):
67         """ Called whenever the user presses a key. """
68         if key == arcade.key.LEFT:
69             self.ball.change_x = -MOVEMENT_SPEED
70         elif key == arcade.key.RIGHT:
71             self.ball.change_x = MOVEMENT_SPEED
72         elif key == arcade.key.UP:
73             self.ball.change_y = MOVEMENT_SPEED
74         elif key == arcade.key.DOWN:
75             self.ball.change_y = -MOVEMENT_SPEED
76
77     def on_key_release(self, key, modifiers):
78         """ Called whenever a user releases a key. """
79         if key == arcade.key.LEFT or key == arcade.key.RIGHT:
80             self.ball.change_x = 0
81         elif key == arcade.key.UP or key == arcade.key.DOWN:
82             self.ball.change_y = 0
83
84
85 def main():
86     window = MyGame(640, 480, "Drawing Example")
87     arcade.run()
88
89
90 main()

```

1.16.4 Moving with the Game Controller

Working with game controllers is a bit more complex. A computer might not have any game controllers, or it might have five controllers plugged in.

We can get a list of all game pads that are plugged in with the `get_joysticks` function. This will either return a list, or it will return nothing at all if there are no game pads.

Below is a block of code that can be put in an `__init__` method for your application that will create an instance variable to represent a game pad if one exists.

```

joysticks = arcade.get_joysticks()
if joysticks:
    self.joystick = joysticks[0]
    self.joystick.open()
else:
    print("There are no joysticks.")
    self.joystick = None

```

1.16.5 Joystick Values

After this, you can get the position of the game controller joystick by calling `self.joystick.x` and `self.joystick.y`.

Try this, combined with the initialization code from above:

Listing 69: move_with_game_controller_print.py

```
1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5 MOVEMENT_SPEED = 5
6 DEAD_ZONE = 0.02
7
8
9 class Ball:
10     def __init__(self, position_x, position_y, change_x, change_y, radius, color):
11
12         # Take the parameters of the init function above, and create instance_
13         #variables out of them.
14         self.position_x = position_x
15         self.position_y = position_y
16         self.change_x = change_x
17         self.change_y = change_y
18         self.radius = radius
19         self.color = color
20
21     def draw(self):
22         """ Draw the balls with the instance variables we have. """
23         arcade.draw_circle_filled(self.position_x, self.position_y, self.radius, self.
24         color)
25
26     def update(self):
27         # Move the ball
28         self.position_y += self.change_y
29         self.position_x += self.change_x
30
31         # See if the ball hit the edge of the screen. If so, change direction
32         if self.position_x < self.radius:
33             self.position_x = self.radius
34
35         if self.position_x > SCREEN_WIDTH - self.radius:
36             self.position_x = SCREEN_WIDTH - self.radius
37
38         if self.position_y < self.radius:
39             self.position_y = self.radius
40
41         if self.position_y > SCREEN_HEIGHT - self.radius:
42             self.position_y = SCREEN_HEIGHT - self.radius
43
44
45 class MyGame(arcade.Window):
46
47     def __init__(self, width, height, title):
48
49         # Call the parent class's init function
50         super().__init__(width, height, title)
51
52         # Make the mouse disappear when it is over the window.
53         # So we just see our object, not the pointer.
54         self.set_mouse_visible(False)
```

(continues on next page)

(continued from previous page)

```

54     arcade.set_background_color(arcade.color.ASH_GREY)
55
56     # Create our ball
57     self.ball = Ball(50, 50, 0, 0, 15, arcade.color.AUBURN)
58
59     # Get a list of all the game controllers that are plugged in
60     joysticks = arcade.get_joysticks()
61
62     # If we have a game controller plugged in, grab it and
63     # make an instance variable out of it.
64     if joysticks:
65         self.joystick = joysticks[0]
66         self.joystick.open()
67     else:
68         print("There are no joysticks.")
69         self.joystick = None
70
71     def on_draw(self):
72
73         """ Called whenever we need to draw the window. """
74         arcade.start_render()
75         self.ball.draw()
76
77     def update(self, delta_time):
78
79         # Update the position according to the game controller
80         if self.joystick:
81             print(self.joystick.x, self.joystick.y)
82
83
84     def main():
85         window = MyGame(640, 480, "Drawing Example")
86         arcade.run()
87
88
89     main()

```

Run the program and see the values it prints out for your game controller as you move the joystick on it around.

- The values will be between -1 and +1, with 0 being a centered joystick.
- The x-axis numbers will be negative if the stick goes left, positive for right.
- The y-axis numbers will be opposite of what you might expect. Negative for up, positive for down.

We can move the ball by adding the following code to the update:

```

def update(self, delta_time):
    # Update the position according to the game controller
    if self.joystick:
        print(self.joystick.x, self.joystick.y)

        self.ball.change_x = self.joystick.x
        self.ball.change_y = -self.joystick.y

```

Notice the – we put in front of setting the y vector. If we don’t do this, the ball will move opposite of what we expect when going up/down. This is because the joystick has y values mapped opposite of how we’d normally expect. There’s a long story to that, which I will not bore you with now.



Fig. 23: Centered (0, 0)



Fig. 24: Down (0, 1)



Fig. 25: Down/Left (-1, 1)



Fig. 26: Down/Right (1, 1)



Fig. 27: Up (0, -1)



Fig. 28: Up/Left (-1, -1)



Fig. 29: Up/Right (1, -1)

But with this code our ball moves *so slow*. How do we speed it up? We can make it run five times faster by multiplying by five if we want.

```
def update(self, delta_time):
    # Update the position according to the game controller
    if self.joystick:
        print(self.joystick.x, self.joystick.y)

    self.ball.change_x = self.joystick.x * 5
    self.ball.change_y = -self.joystick.y * 5
```

Or better yet, define a constant variable at the top of your program and use that. In our final example below, we'll do just that.

1.16.6 Deadzone

What if your ball ‘drifts’ when you have the joystick centered?

Joysticks are mechanical. A centered joystick might have a value not at 0, but at 0.0001 or some small number. This will make for a small “drift” on a person’s character. We often counteract this by having a “dead zone” where if the number is below a certain value, we just assume it is zero to eliminate the drift.

See the highlighted lines for how we take care of the dead zone:

Listing 70: move_with_game_controller.py

```
1 import arcade
2
3 SCREEN_WIDTH = 640
4 SCREEN_HEIGHT = 480
5 MOVEMENT_SPEED = 5
6 DEAD_ZONE = 0.02
7
8
9 class Ball:
10     def __init__(self, position_x, position_y, change_x, change_y, radius, color):
```

(continues on next page)

(continued from previous page)

```

12     # Take the parameters of the init function above, and create instance_
13     ↪variables out of them.
14         self.position_x = position_x
15         self.position_y = position_y
16         self.change_x = change_x
17         self.change_y = change_y
18         self.radius = radius
19         self.color = color
20
21     def draw(self):
22         """ Draw the balls with the instance variables we have. """
23         arcade.draw_circle_filled(self.position_x, self.position_y, self.radius,
24         ↪color)
25
26     def update(self):
27         # Move the ball
28         self.position_y += self.change_y
29         self.position_x += self.change_x
30
31         # See if the ball hit the edge of the screen. If so, change direction
32         if self.position_x < self.radius:
33             self.position_x = self.radius
34
35         if self.position_x > SCREEN_WIDTH - self.radius:
36             self.position_x = SCREEN_WIDTH - self.radius
37
38         if self.position_y < self.radius:
39             self.position_y = self.radius
40
41         if self.position_y > SCREEN_HEIGHT - self.radius:
42             self.position_y = SCREEN_HEIGHT - self.radius
43
44 class MyGame(arcade.Window):
45
46     def __init__(self, width, height, title):
47
48         # Call the parent class's init function
49         super().__init__(width, height, title)
50
51         # Make the mouse disappear when it is over the window.
52         # So we just see our object, not the pointer.
53         self.set_mouse_visible(False)
54
55         arcade.set_background_color(arcade.color.ASH_GREY)
56
57         # Create our ball
58         self.ball = Ball(50, 50, 0, 0, 15, arcade.color.AUBURN)
59
60         # Get a list of all the game controllers that are plugged in
61         joysticks = arcade.get_joysticks()
62
63         # If we have a game controller plugged in, grab it and
64         # make an instance variable out of it.
65         if joysticks:
66             self.joystick = joysticks[0]
67             self.joystick.open()

```

(continues on next page)

(continued from previous page)

```

67     else:
68         print("There are no joysticks.")
69         self.joystick = None
70
71     def on_draw(self):
72
73         """ Called whenever we need to draw the window. """
74         arcade.start_render()
75         self.ball.draw()
76
77     def update(self, delta_time):
78
79         # Update the position according to the game controller
80         if self.joystick:
81
82             # Set a "dead zone" to prevent drive from a centered joystick
83             if abs(self.joystick.x) < DEAD_ZONE:
84                 self.ball.change_x = 0
85             else:
86                 self.ball.change_x = self.joystick.x * MOVEMENT_SPEED
87
88             # Set a "dead zone" to prevent drive from a centered joystick
89             if abs(self.joystick.y) < DEAD_ZONE:
90                 self.ball.change_y = 0
91             else:
92                 self.ball.change_y = -self.joystick.y * MOVEMENT_SPEED
93
94         self.ball.update()
95
96
97     def main():
98         window = MyGame(640, 480, "Drawing Example")
99         arcade.run()
100
101
102 main()

```

1.17 Sound Effects

Adding sound to your game isn't too hard. There are two steps:

1. Load the sound
2. Play the sound

For these examples, I'm using a sound file called `laser.ogg` that you can download [here](#). Make sure you save the file in the same directory as any Python program that tries to use it.

1.17.1 Loading Sounds

The code below creates a new variable called `laser_sound`. It calls arcades `load_sound` function. It passes the filename of our sound.

```
laser_sound = arcade.load_sound("laser.ogg")
```

For this to work, you need to have a sound downloaded and named `laser.ogg` in the same directory as your Python file. The computer will not find the sound if it is in a different directory.

This loads the sound, but does not play it. We only want to load the sound once. We don't want to load the sound off the disk every time we play it. It can be kept in memory.

1.17.2 Playing Sounds

The code to play sounds is straight-forward. Just call the `play_sound` function, and pass in the variable that we set equal to the sound we loaded:

```
arcade.play_sound(laser_sound)
```

Putting the two together, you might think we could do this to play sounds:

```
import arcade

laser_sound = arcade.load_sound("laser.ogg")
arcade.play_sound(laser_sound)
```

But that doesn't work. The program ends before the sound has a chance to play. The `play_sound` button doesn't pause while the sound plays, it returns right away and keeps going. This was a similar issue that we had when we opened a window, and we can solve it the same way:

```
1 import arcade
2
3 arcade.open_window(300, 300, "Sound Demo")
4 laser_sound = arcade.load_sound("laser.ogg")
5 arcade.play_sound(laser_sound)
6 arcade.run()
```

For this reason, I recommend loading the sound in the `__init__` method of the class that will play the sound.

1.17.3 Triggering Sounds

We want to play the sound when something happens. So this example loads the sound once during the `__init__`. When the user hits the space bar, that is when we trigger the sound to play.

```
1 import arcade
2
3
4 class MyApplication(arcade.Window):
5
6     def __init__(self, width, height):
7         super().__init__(width, height, "Trigger Sound With Key")
8
9         # Load the sound when the application starts
10        self.laser_sound = arcade.load_sound("laser.ogg")
11
12    def on_key_press(self, key, modifiers):
13
14        # If the user hits the space bar, play the sound that we loaded
```

(continues on next page)

(continued from previous page)

```
15     if key == arcade.key.SPACE:
16         arcade.play_sound(self.laser_sound)
17
18
19 def main():
20     window = MyApplication(300, 300)
21     arcade.run()
22
23 main()
```

1.17.4 Finding Sounds

Great places to find free sounds to use in your program:

- [OpenGameArt.org](#)
- [www.freesound.org](#) is ok, but requires a login, which is a pain.
- [Kenney.nl](#) has a few free sound packs you can download, and several that are cheap.

1.17.5 Sound File Formats

There are several types of [sound file formats](#) that you can find sounds in:

- .wav - This is a raw “wave” format. The sound is not compressed at all. You do not need a special library to decompress the sound, but the sound file itself can be rather large.
- .mp3 - MPEG Layer III Audio (mp3) is one of the most popular compressed sound file formats. This file format is what enabled on-line music to become popular. However some of the compression algorithms have patents on them, making it not as suitable for free software.
- .m4a - Apple’s file format for compressed, but unprotected audio.
- .ogg - A open-source sound file that uses Ogg-Vorbis for compression. A less popular but patent-free method of sound storage.

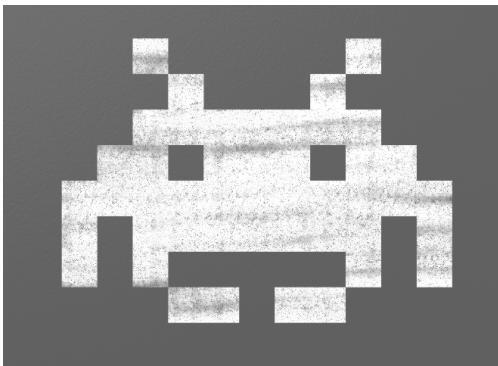
Arcade should be able to play files in either the mp3 or ogg format. If you have issues getting it to work, try converting the sound to a raw wav format.

If you need to convert file formats, or trim a sound file, I suggest using the program [Audacity](#).

1.18 Sprites And Collisions

Our games need support for handling objects that collide. Balls bouncing off paddles, laser beams hitting aliens, or our favorite character collecting a coin. All these examples require collision detection.

The Arcade library has support for sprites. A sprite is a two-dimensional image that is part of the larger graphical scene. Typically a sprite will be some kind of object in the scene that will be interacted with like a car, frog, or little plumber guy.



Originally, video game consoles had built-in hardware support for sprites. Now this specialized hardware support is no longer needed, but we still use the term “sprite.” The [history of sprites](#) is interesting, if you want to read up more about it.

1.18.1 Finding Images for Sprites

There are several image formats that computers use:

- .bmp - Bitmap. This is an uncompressed image that normally uses three bytes to represent each dot in the image. These files can be very large. Because there are so many better options, this format is not used often. It is, however, the simplest format to use.
- .png - Great patent-free format for line art and clip art. Not great for photos. Can't hold animations.
- .gif - Great format for line art and clip art. Has had issues with patents (now expired). Can do animations.
- .jpg - Great file format for photos. Terrible for clip-art. Don't use for sprites.
- .svg - File format for storing line-art images that can scale to any resolution. Not compatible with the “arcade” library.

If you use Google’s [advanced image search](#) you can find images that are “icon” sized, and either png or gif file format.

There’s also a great source for images from [kenney.nl](#). He has a lot of free and cheap game image assets. That’s where the following images come from that we will use in our examples:



Fig. 30: character.png

Where to Save Images

Where should you save them? If you load your sprite with the code below, the computer will look for the character.png image in the same directory as your Python file. Save the image anywhere else, and it won’t be found.



Fig. 31: coin_01.png

How to Reference Images

If you create your own game that you publish, you need to:

- Create your own images
- Hire someone to create your images
- Buy your own images with a license to use them in your own game
- Find images that are public domain or licensed for public use

If you are just creating a game for class that won't be used in public, then right before you load the image leave a comment with the source. I'll show this in a bit.

Attention: Do not list "Google" as a source. That's like using "The Library" as a source in your report. Find the source of the image that Google is pointing to.

1.18.2 Basic Sprites and Collisions

Let's step through an example program that uses sprites. This example shows how to create a screen of sprites that are coins, and collect them using a sprite that is a character image controlled by the mouse as shown in the figure below. The program keeps "score" on how many coins have been collected. The code for this example may be found at:

http://arcade.academy/examples/sprite_collect_coins.html

In this chapter, we'll step through that example.

Fig. 32: Example Sprite Game

Getting the Application Started

The first few lines of our program start off like other games we've done. We import a couple libraries. Set a couple constants for the size of the screen, and a couple new constants that we will use to scale our graphics.

The example below should have nothing new, it just creates a window and sets a background color. We'll add in the new stuff soon.

Listing 71: Sprite Sample Start

```

1  """ Sprite Sample Program """
2
3 import random
4 import arcade
5
6 # --- Constants ---
7 SPRITE_SCALING_PLAYER = 0.5
8 SPRITE_SCALING_COIN = 0.2
9 COIN_COUNT = 50
10
11 SCREEN_WIDTH = 800
12 SCREEN_HEIGHT = 600
13
14
15 class MyGame(arcade.Window):
16     """ Our custom Window Class"""
17
18     def __init__(self):
19         """ Initializer """
20         # Call the parent class initializer
21         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
22
23     def on_draw(self):
24         arcade.start_render()
25
26
27 def main():
28     """ Main method """
29     window = MyGame()
30     arcade.run()
31
32
33 if __name__ == "__main__":
34     main()

```

The Constructor

What's next? We need to add our attributes to the `MyGame` class. We add our attributes to the `__init__` method. Here is our code with the expanded `__init__`:

Listing 72: Expanded Init

```

1  """ Sprite Sample Program """
2
3 import random
4 import arcade
5
6 # --- Constants ---
7 SPRITE_SCALING_PLAYER = 0.5
8 SPRITE_SCALING_COIN = 0.2
9 COIN_COUNT = 50
10
11 SCREEN_WIDTH = 800

```

(continues on next page)

(continued from previous page)

```

12 SCREEN_HEIGHT = 600
13
14
15 class MyGame(arcade.Window):
16     """ Our custom Window Class"""
17
18     def __init__(self):
19         """ Initializer """
20         # Call the parent class initializer
21         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
22
23         # Variables that will hold sprite lists.
24         self.player_list = None
25         self.coin_list = None
26
27         # Set up the player info
28         self.player_sprite = None
29         self.score = 0
30
31         # Don't show the mouse cursor
32         self.set_mouse_visible(False)
33
34         arcade.set_background_color(arcade.color.AMAZON)
35
36     def on_draw(self):
37         arcade.start_render()
38
39
40     def main():
41         """ Main method """
42         window = MyGame()
43         arcade.run()
44
45
46 if __name__ == "__main__":
47     main()

```

The variables we are creating:

- `player_list`: When working with sprites, we normally put them into lists. Other game engines might call these sprite groups, or sprite layers. Our game will have one list for the player, and one list for the coins. Even if there is only one sprite, we should still put it in a list because there is a lot of code in `SpriteList` to optimize drawing.
- `coin_list`: This is a list of all the coins. We will be checking if the player touches any sprite in this list.
- `player_sprite`: This points to our player's sprite. It is the sprite we will move.
- `score`: This keeps track of our score.

We use a command built into the parent `Window` class called `set_mouse_visible` to make the mouse not visible. Finally we set the background color.

The `setup` Function

Next up, we will create a `setup` method. This will create our sprites and get our game set up. We do this in a different method than `__init__` so that if we ever want to restart the game, we can just call `setup` again.

The `setup` method is not called automatically. Therefore in the example below, note we have added the code that calls the `setup` function near the end: `window.setup()`.

Listing 73: Sprite Sample With Player

```

1  """ Sprite Sample Program """
2
3 import random
4 import arcade
5
6 # --- Constants ---
7 SPRITE_SCALING_PLAYER = 0.5
8 SPRITE_SCALING_COIN = 0.2
9 COIN_COUNT = 50
10
11 SCREEN_WIDTH = 800
12 SCREEN_HEIGHT = 600
13
14
15 class MyGame(arcade.Window):
16     """ Our custom Window Class"""
17
18     def __init__(self):
19         """ Initializer """
20         # Call the parent class initializer
21         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
22
23         # Variables that will hold sprite lists
24         self.player_list = None
25         self.coin_list = None
26
27         # Set up the player info
28         self.player_sprite = None
29         self.score = 0
30
31         # Don't show the mouse cursor
32         self.set_mouse_visible(False)
33
34         arcade.set_background_color(arcade.color.AMAZON)
35
36     def setup(self):
37         """ Set up the game and initialize the variables. """
38
39         # Sprite lists
40         self.player_list = arcade.SpriteList()
41         self.coin_list = arcade.SpriteList()
42
43         # Score
44         self.score = 0
45
46         # Set up the player
47         # Character image from kenney.nl
48         self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
49         self.player_sprite.center_x = 50
50         self.player_sprite.center_y = 50
51         self.player_list.append(self.player_sprite)
52
53     def on_draw(self):

```

(continues on next page)

(continued from previous page)

```

54     arcade.start_render()

55
56     # Draw the sprite lists here. Typically sprites are divided into
57     # different groups. Other game engines might call these "sprite layers"
58     # or "sprite groups." Sprites that don't move should be drawn in their
59     # own group for the best performance, as Arcade can tell the graphics
60     # card to just redraw them at the same spot.
61     # Try to avoid drawing sprites on their own, use a SpriteList
62     # because there are many performance improvements in that code.
63     self.coin_list.draw()
64     self.player_list.draw()

65
66
67 def main():
68     """ Main method """
69     window = MyGame()
70     window.setup()
71     arcade.run()

72
73
74 if __name__ == "__main__":
75     main()

```

How does the code above work?

First, we need some lists to hold our sprites. We could do use a list like this:

```
coin_list = []
```

But wait! `coin_list` is an instance variable that's part of our class. we need to prepend it with `self..`

```
self.coin_list = []
```

However, the Arcade library has a class especially for handling sprite lists. This class is called `SpriteList`. For more information, check out the [SpriteList](#) documentation. So instead of creating an empty list with `[]`, we will create a new instance of the `SpriteList` class:

```
self.coin_list = SpriteList()
```

Except that doesn't work. Why? `SpriteList` is in the Arcade library. We need to prepend any reference to things in the Arcade library with `arcade` of course, so now we have:

```
self.coin_list = arcade.SpriteList()
```

We need a separate list for just coins. This list won't have the player. We also need to reset our score to 0.

```
self.coin_list = arcade.SpriteList()

self.score = 0
```

Now we need to create our sprites. The name of the class that represents sprites is called `Sprite`. You can read more about it by looking at the [Sprite](#) documentation. The `Sprite` constructor takes two parameters. A path to the image we will be using, and how big to scale it.

For class, please source the image right before you load it. If you drew your own image, please note that as well.

```
# Character image from kenney.nl
self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
```

How do we draw all our sprites? Really easy. Just call the `draw` method that exists for us in the `SpriteList` class. We just need to do this for each of our sprite lists.

```
def on_draw(self):
    arcade.start_render()

    # Draw all the sprite lists.
    self.coin_list.draw()
    self.player_list.draw()
```

Wait. We don't have many sprites. There are no coins, and we have just the player. Let's add a `for` loop to our program and create a bunch of coins:

Listing 74: Sprite Sample With Player And Coins

```
1  """ Sprite Sample Program """
2
3  import random
4  import arcade
5
6  # --- Constants ---
7  SPRITE_SCALING_PLAYER = 0.5
8  SPRITE_SCALING_COIN = 0.2
9  COIN_COUNT = 50
10
11 SCREEN_WIDTH = 800
12 SCREEN_HEIGHT = 600
13
14
15 class MyGame(arcade.Window):
16     """ Our custom Window Class"""
17
18     def __init__(self):
19         """ Initializer """
20
21         # Call the parent class initializer
22         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
23
24         # Variables that will hold sprite lists
25         self.coin_list = None
26         self.player_list = None
27
28         # Set up the player info
29         self.player_sprite = None
30         self.score = 0
31
32         # Don't show the mouse cursor
33         self.set_mouse_visible(False)
34
35         arcade.set_background_color(arcade.color.AMAZON)
36
37     def setup(self):
38         """ Set up the game and initialize the variables. """
39
```

(continues on next page)

(continued from previous page)

```

39     # Sprite lists
40     self.player_list = arcade.SpriteList()
41     self.coin_list = arcade.SpriteList()
42
43     # Score
44     self.score = 0
45
46     # Set up the player
47     # Character image from kenney.nl
48     self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
49     self.player_sprite.center_x = 50
50     self.player_sprite.center_y = 50
51     self.player_list.append(self.player_sprite)
52
53     # Create the coins
54     for i in range(COIN_COUNT):
55
56         # Create the coin instance
57         # Coin image from kenney.nl
58         coin = arcade.Sprite("coin_01.png", SPRITE_SCALING_COIN)
59
60         # Position the coin
61         coin.center_x = random.randrange(SCREEN_WIDTH)
62         coin.center_y = random.randrange(SCREEN_HEIGHT)
63
64         # Add the coin to the lists
65         self.coin_list.append(coin)
66
67     def on_draw(self):
68         """ Draw everything """
69         arcade.start_render()
70
71         # Draw the sprite lists here. Typically sprites are divided into
72         # different groups. Other game engines might call these "sprite layers"
73         # or "sprite groups." Sprites that don't move should be drawn in their
74         # own group for the best performance, as Arcade can tell the graphics
75         # card to just redraw them at the same spot.
76         # Try to avoid drawing sprites on their own, and not in a layer.
77         # There are many performance improvements to drawing in a layer.
78         self.coin_list.draw()
79         self.player_list.draw()
80
81
82     def main():
83         """ Main method """
84         window = MyGame()
85         window.setup()
86         arcade.run()
87
88
89     if __name__ == "__main__":
90         main()

```

Drawing The Score

In addition to drawing the sprites, let's go ahead and put the score on the screen:

```
# Put the text on the screen.
output = "Score: " + str(self.score)
arcade.draw_text(output, 10, 20, arcade.color.WHITE, 14)
```

Rather than do that "Score: " + str(self.score) it is possible to do print formatting if you are using Python 3.6 or later. We'll talk more about print formatting later, but that code would look like:

```
# Put the text on the screen.
output = f"Score: {self.score}"
arcade.draw_text(output, 10, 20, arcade.color.WHITE, 14)
```

There are three standards for how to format strings in Python, so that whole subject is a bit confusing.

The On Mouse Motion Method

Moving the player sprite with the mouse is easy. All sprites have instance variables `center_x` and `center_y`. Just change those values to the mouse's x and y location to move the sprite.

```
def on_mouse_motion(self, x, y, dx, dy):
    self.player_sprite.center_x = x
    self.player_sprite.center_y = y
```

Now, our whole program looks like:

Listing 75: Sprite Sample With Mouse Motion And Score

```
""" Sprite Sample Program """

import random
import arcade

# --- Constants ---
SPRITE_SCALING_PLAYER = 0.5
SPRITE_SCALING_COIN = 0.2
COIN_COUNT = 50

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600

class MyGame(arcade.Window):
    """ Our custom Window Class"""

    def __init__(self):
        """Initializer"""
        # Call the parent class initializer
        super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")

        # Variables that will hold sprite lists
        self.player_list = None
        self.coin_list = None

        # Set up the player info
        self.player_sprite = None
        self.score = 0
```

(continues on next page)

(continued from previous page)

```

30
31     # Don't show the mouse cursor
32     self.set_mouse_visible(False)
33
34     arcade.set_background_color(arcade.color.AMAZON)
35
36 def setup(self):
37     """ Set up the game and initialize the variables. """
38
39     # Sprite lists
40     self.player_list = arcade.SpriteList()
41     self.coin_list = arcade.SpriteList()
42
43     # Score
44     self.score = 0
45
46     # Set up the player
47     # Character image from kenney.nl
48     self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
49     self.player_sprite.center_x = 50
50     self.player_sprite.center_y = 50
51     self.player_list.append(self.player_sprite)
52
53     # Create the coins
54     for i in range(COIN_COUNT):
55
56         # Create the coin instance
57         # Coin image from kenney.nl
58         coin = arcade.Sprite("coin_01.png", SPRITE_SCALING_COIN)
59
60         # Position the coin
61         coin.center_x = random.randrange(SCREEN_WIDTH)
62         coin.center_y = random.randrange(SCREEN_HEIGHT)
63
64         # Add the coin to the lists
65         self.coin_list.append(coin)
66
67 def on_draw(self):
68     """ Draw everything """
69     arcade.start_render()
70     self.coin_list.draw()
71     self.player_list.draw()
72
73     # Put the text on the screen.
74     output = f"Score: {self.score}"
75     arcade.draw_text(output, 10, 20, arcade.color.WHITE, 14)
76
77 def on_mouse_motion(self, x, y, dx, dy):
78     """ Handle Mouse Motion """
79
80     # Move the center of the player sprite to match the mouse x, y
81     self.player_sprite.center_x = x
82     self.player_sprite.center_y = y
83
84
85 def main():
86     """ Main method """

```

(continues on next page)

(continued from previous page)

```

87     window = MyGame()
88     window.setup()
89     arcade.run()
90
91
92 if __name__ == "__main__":
93     main()

```

The Update Method

Our update method needs to do three things:

1. Update each of the sprites
2. Check to see if the player is touching any coins
3. Remove any coins colliding with the player, and update the score.

Each sprite has its own update method. This allows sprites to move and animate its images. Right now, our sprite does not have this method. But we will soon. Rather than call the update method of each sprite we have, there is an update method in each sprite list that will call update on each sprite in the list. Therefore, just calling update with our coin_list will cause all coin sprites to update.

```
self.coin_list.update()
```

How do we detect what coins are touching the player? We call the check_for_collision_with_list method. Pass it in our player sprite, along with a list of all the coins. That function will return a list of all colliding sprites. If no sprites collide, the list will be empty.

```
# Generate a list of all sprites that collided with the player.
coins_hit_list = arcade.check_for_collision_with_list(self.player_sprite, self.coin_
list)
```

What do we do with this hit_list we get back? We loop through it. We add one to the score for each sprite hit.

We also need to get rid of the sprite. The sprite class has a method called kill. This method will remove the sprite from existence.

```
# Loop through each colliding sprite, remove it, and add to the score.
for coin in coins_hit_list:
    coin.kill()
    self.score += 1
```

Here's the whole update method put together:

Listing 76: Sprite Sample With Update Method

```

1 """ Sprite Sample Program """
2
3 import random
4 import arcade
5
6 # --- Constants ---
7 SPRITE_SCALING_PLAYER = 0.5
8 SPRITE_SCALING_COIN = 0.2
9 COIN_COUNT = 50

```

(continues on next page)

(continued from previous page)

```

10 SCREEN_WIDTH = 800
11 SCREEN_HEIGHT = 600
12
13
14
15 class MyGame(arcade.Window):
16     """ Our custom Window Class"""
17
18     def __init__(self):
19         """ Initializer """
20         # Call the parent class initializer
21         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
22
23         # Variables that will hold sprite lists
24         self.player_list = None
25         self.coin_list = None
26
27         # Set up the player info
28         self.player_sprite = None
29         self.score = 0
30
31         # Don't show the mouse cursor
32         self.set_mouse_visible(False)
33
34         arcade.set_background_color(arcade.color.AMAZON)
35
36     def setup(self):
37         """ Set up the game and initialize the variables. """
38
39         # Sprite lists
40         self.player_list = arcade.SpriteList()
41         self.coin_list = arcade.SpriteList()
42
43         # Score
44         self.score = 0
45
46         # Set up the player
47         # Character image from kenney.nl
48         self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
49         self.player_sprite.center_x = 50
50         self.player_sprite.center_y = 50
51         self.player_list.append(self.player_sprite)
52
53         # Create the coins
54         for i in range(COIN_COUNT):
55
56             # Create the coin instance
57             # Coin image from kenney.nl
58             coin = arcade.Sprite("coin_01.png", SPRITE_SCALING_COIN)
59
60             # Position the coin
61             coin.center_x = random.randrange(SCREEN_WIDTH)
62             coin.center_y = random.randrange(SCREEN_HEIGHT)
63
64             # Add the coin to the lists
65             self.coin_list.append(coin)
66

```

(continues on next page)

(continued from previous page)

```

67     def on_draw(self):
68         """ Draw everything """
69         arcade.start_render()
70         self.coin_list.draw()
71         self.player_list.draw()
72
73         # Put the text on the screen.
74         output = f"Score: {self.score}"
75         arcade.draw_text(output, 10, 20, arcade.color.WHITE, 14)
76
77     def on_mouse_motion(self, x, y, dx, dy):
78         """ Handle Mouse Motion """
79
80         # Move the center of the player sprite to match the mouse x, y
81         self.player_sprite.center_x = x
82         self.player_sprite.center_y = y
83
84     def update(self, delta_time):
85         """ Movement and game logic """
86
87         # Call update on all sprites (The sprites don't do much in this
88         # example though.)
89         self.coin_list.update()
90
91         # Generate a list of all sprites that collided with the player.
92         coins_hit_list = arcade.check_for_collision_with_list(self.player_sprite,
93                                         self.coin_list)
94
95         # Loop through each colliding sprite, remove it, and add to the score.
96         for coin in coins_hit_list:
97             coin.kill()
98             self.score += 1
99
100
101    def main():
102        """ Main method """
103        window = MyGame()
104        window.setup()
105        arcade.run()
106
107
108    if __name__ == "__main__":
109        main()

```

1.19 Moving Sprites

How do we get sprites to move?

To customize our sprite's behavior, we need to subclass the `Sprite` class with our own child class. This is easy:

```
class Coin(arcade.Sprite):
```

We need to provide each sprite with a `update` method. The `update` method is automatically called to update the sprite's position.

```
class Coin(arcade.Sprite):  
  
    def update(self):  
        # Code to move goes here
```

Wait! We have a new class called Coin, but we aren't using it. Find in our original code this line:

```
coin = arcade.Sprite("coin_01.png", COIN_SPRITE_SCALING)
```

See how it is creating an instance of Sprite? We want to create an instance of our new Coin class instead:

```
coin = Coin("coin_01.png", COIN_SPRITE_SCALING)
```

Now, how do we get the coin to move?

1.19.1 Moving Sprites Down

To get the sprites to “fall” down the screen, we need to make their y location smaller. This is easy. Over-ride update in the sprite and subtract from y each frame:

```
class Coin(arcade.Sprite):  
  
    def update(self):  
        self.center_y -= 1
```

Next, create an instance of the Coin class instead of a Sprite class.

Listing 77: Sprite Sample Move Down

```
1  """ Sprite Sample Program """  
2  
3  import random  
4  import arcade  
5  
6  # --- Constants ---  
7  SPRITE_SCALING_PLAYER = 0.5  
8  SPRITE_SCALING_COIN = 0.2  
9  COIN_COUNT = 50  
10  
11 SCREEN_WIDTH = 800  
12 SCREEN_HEIGHT = 600  
13  
14  
15 class Coin(arcade.Sprite):  
16  
17     def update(self):  
18         self.center_y -= 1  
19  
20  
21 class MyGame(arcade.Window):  
22     """ Our custom Window Class """  
23  
24     def __init__(self):  
25         """ Initializer """  
26         # Call the parent class initializer  
27         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
```

(continues on next page)

(continued from previous page)

```

28
29     # Variables that will hold sprite lists
30     self.player_list = None
31     self.coin_list = None
32
33     # Set up the player info
34     self.player_sprite = None
35     self.score = 0
36
37     # Don't show the mouse cursor
38     self.set_mouse_visible(False)
39
40     arcade.set_background_color(arcade.color.AMAZON)
41
42     def setup(self):
43         """ Set up the game and initialize the variables. """
44
45         # Sprite lists
46         self.player_list = arcade.SpriteList()
47         self.coin_list = arcade.SpriteList()
48
49         # Score
50         self.score = 0
51
52         # Set up the player
53         # Character image from kenney.nl
54         self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
55         self.player_sprite.center_x = 50
56         self.player_sprite.center_y = 50
57         self.player_list.append(self.player_sprite)
58
59         # Create the coins
60         for i in range(COIN_COUNT):
61
62             # Create the coin instance
63             # Coin image from kenney.nl
64             coin = Coin("coin_01.png", SPRITE_SCALING_COIN)
65
66             # Position the coin
67             coin.center_x = random.randrange(SCREEN_WIDTH)
68             coin.center_y = random.randrange(SCREEN_HEIGHT)
69
70             # Add the coin to the lists
71             self.coin_list.append(coin)
72
73     def on_draw(self):
74         """ Draw everything """
75         arcade.start_render()
76         self.coin_list.draw()
77         self.player_list.draw()
78
79         # Put the text on the screen.
80         output = f"Score: {self.score}"
81         arcade.draw_text(output, 10, 20, arcade.color.WHITE, 14)
82
83     def on_mouse_motion(self, x, y, dx, dy):
84         """ Handle Mouse Motion """

```

(continues on next page)

(continued from previous page)

```

85
86     # Move the center of the player sprite to match the mouse x, y
87     self.player_sprite.center_x = x
88     self.player_sprite.center_y = y
89
90     def update(self, delta_time):
91         """ Movement and game logic """
92
93         # Call update on all sprites (The sprites don't do much in this
94         # example though.)
95         self.coin_list.update()
96
97         # Generate a list of all sprites that collided with the player.
98         hit_list = arcade.check_for_collision_with_list(self.player_sprite,
99                                            self.coin_list)
100
101        # Loop through each colliding sprite, remove it, and add to the score.
102        for coin in hit_list:
103            coin.kill()
104            self.score += 1
105
106
107    def main():
108        """ Main method """
109        window = MyGame()
110        window.setup()
111        arcade.run()
112
113
114    if __name__ == "__main__":
115        main()

```

This causes the coins to move down. But once they move off the screen they keep going into negative-coordinate land. We can't see them any more. Sad.

Fig. 33: Coins moving down

Resetting to the Top

We can get around this by resetting the coins up to the top. Here's how its done:

```

class Coin(arcade.Sprite):

    def update(self):
        self.center_y -= 1

        # See if we went off-screen
        if self.center_y < 0:
            self.center_y = SCREEN_HEIGHT

```

But this isn't perfect. Because if your eyes are fast, you can see the coin 'pop' in and out of existence at the edge. It doesn't smoothly slide off. This is because we move it when the *center* of the coin is at the edge. Not the top of the coin has slid off.

There are a couple ways we can do this. Here's one. We'll check at -20 instead of 0. As long as the coin radius is 20 or less, we are good.

```
class Coin(arcade.Sprite):

    def update(self):
        self.center_y -= 1

        # See if we went off-screen
        if self.center_y < -20:
            self.center_y = SCREEN_HEIGHT + 20
```

There's another way. In addition to `center_y`, sprites have other members that are useful in these cases. They are `top`, `bottom`, `left` and `right`. So we can do this:

```
class Coin(arcade.Sprite):

    def update(self):
        self.center_y -= 1

        # See if we went off-screen
        if self.top < 0:
            self.bottom = SCREEN_HEIGHT
```

Doing this allows the coins to smoothly slide on and off the screen. But since they reappear at the top, we get repeating patterns. See the image below:

Fig. 34: Coins repeating in a pattern

Instead we can randomize it a bit:

```
def update(self):

    # Move the coin
    self.center_y -= 1

    # See if the coin has fallen off the bottom of the screen.
    # If so, reset it.
    if self.top < 0:
        # Reset the coin to a random spot above the screen
        self.center_y = random.randrange(SCREEN_HEIGHT + 20,
                                         SCREEN_HEIGHT + 100)
        self.center_x = random.randrange(SCREEN_WIDTH)
```

Never Ending Coins

This works, but when we collect all the coins we are done. What if it was a never-ending set of coins? Instead of “killing” the coin, let's reset it to the top of the screen.

```
def update(self, delta_time):
    """ Movement and game logic """

    self.coin_list.update()

    hit_list = arcade.check_for_collision_with_list(self.player_sprite, self.coin_
list)
```

(continues on next page)

(continued from previous page)

```

for coin in hit_list:
    self.score += 1

    # Reset the coin to a random spot above the screen
    coin.center_y = random.randrange(SCREEN_HEIGHT + 20,
                                      SCREEN_HEIGHT + 100)
    coin.center_x = random.randrange(SCREEN_WIDTH)

```

We can even take that common code, and move it to a method. Here's a full example:

Listing 78: Full Move Down Sprite Sample

```

1  """ Sprite Sample Program """
2
3 import random
4 import arcade
5
6 # --- Constants ---
7 SPRITE_SCALING_PLAYER = 0.5
8 SPRITE_SCALING_COIN = 0.2
9 COIN_COUNT = 50
10
11 SCREEN_WIDTH = 800
12 SCREEN_HEIGHT = 600
13
14
15 class Coin(arcade.Sprite):
16     """
17         This class represents the coins on our screen. It is a child class of
18         the arcade library's "Sprite" class.
19     """
20
21     def reset_pos(self):
22
23         # Reset the coin to a random spot above the screen
24         self.center_y = random.randrange(SCREEN_HEIGHT + 20,
25                                         SCREEN_HEIGHT + 100)
26         self.center_x = random.randrange(SCREEN_WIDTH)
27
28     def update(self):
29
30         # Move the coin
31         self.center_y -= 1
32
33         # See if the coin has fallen off the bottom of the screen.
34         # If so, reset it.
35         if self.top < 0:
36             self.reset_pos()
37
38
39 class MyGame(arcade.Window):
40     """
41         Our custom Window Class"""
42
43     def __init__(self):
44         """
45             Initializer """
46         # Call the parent class initializer

```

(continues on next page)

(continued from previous page)

```

45     super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
46
47     # Variables that will hold sprite lists
48     self.player_list = None
49     self.coin_list = None
50
51     # Set up the player info
52     self.player_sprite = None
53     self.score = 0
54
55     # Don't show the mouse cursor
56     self.set_mouse_visible(False)
57
58     arcade.set_background_color(arcade.color.AMAZON)
59
60     def setup(self):
61         """ Set up the game and initialize the variables. """
62
63         # Sprite lists
64         self.player_list = arcade.SpriteList()
65         self.coin_list = arcade.SpriteList()
66
67         # Score
68         self.score = 0
69
70         # Set up the player
71         # Character image from kenney.nl
72         self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
73         self.player_sprite.center_x = 50
74         self.player_sprite.center_y = 50
75         self.player_list.append(self.player_sprite)
76
77         # Create the coins
78         for i in range(COIN_COUNT):
79
80             # Create the coin instance
81             # Coin image from kenney.nl
82             coin = Coin("coin_01.png", SPRITE_SCALING_COIN)
83
84             # Position the coin
85             coin.center_x = random.randrange(SCREEN_WIDTH)
86             coin.center_y = random.randrange(SCREEN_HEIGHT)
87
88             # Add the coin to the lists
89             self.coin_list.append(coin)
90
91     def on_draw(self):
92         """ Draw everything """
93         arcade.start_render()
94         self.coin_list.draw()
95         self.player_list.draw()
96
97         # Put the text on the screen.
98         output = f"Score: {self.score}"
99         arcade.draw_text(output, 10, 20, arcade.color.WHITE, 14)
100
101    def on_mouse_motion(self, x, y, dx, dy):

```

(continues on next page)

(continued from previous page)

```

102     """ Handle Mouse Motion """
103
104     # Move the center of the player sprite to match the mouse x, y
105     self.player_sprite.center_x = x
106     self.player_sprite.center_y = y
107
108     def update(self, delta_time):
109         """ Movement and game logic """
110
111         # Call update on all sprites (The sprites don't do much in this
112         # example though.)
113         self.coin_list.update()
114
115         # Generate a list of all sprites that collided with the player.
116         hit_list = arcade.check_for_collision_with_list(self.player_sprite,
117                                                       self.coin_list)
118
119         # Loop through each colliding sprite, remove it, and add to the score.
120         for coin in hit_list:
121             coin.kill()
122             self.score += 1
123
124
125     def main():
126         """ Main method """
127         window = MyGame()
128         window.setup()
129         arcade.run()
130
131
132     if __name__ == "__main__":
133         main()
134

```

1.19.2 Bouncing Coins

Instead of always adding one to the y-coordinate and have the sprites move down, we can keep a vector by using `change_x` and `change_y`. By using these, we can have the sprite bounce around the screen:

Fig. 35: Coins bouncing around

Listing 79: sprites_sample_move_bouncing.py

```

1     """ Sprite Sample Program """
2
3     import random
4     import arcade
5
6     # --- Constants ---
7     SPRITE_SCALING_PLAYER = 0.5
8     SPRITE_SCALING_COIN = 0.2
9     COIN_COUNT = 50
10

```

(continues on next page)

(continued from previous page)

```

11 SCREEN_WIDTH = 800
12 SCREEN_HEIGHT = 600
13
14
15 class Coin(arcade.Sprite):
16
17     def __init__(self, filename, sprite_scaling):
18
19         super().__init__(filename, sprite_scaling)
20
21         self.change_x = 0
22         self.change_y = 0
23
24     def update(self):
25
26         # Move the coin
27         self.center_x += self.change_x
28         self.center_y += self.change_y
29
30         # If we are out-of-bounds, then 'bounce'
31         if self.left < 0:
32             self.change_x *= -1
33
34         if self.right > SCREEN_WIDTH:
35             self.change_x *= -1
36
37         if self.bottom < 0:
38             self.change_y *= -1
39
40         if self.top > SCREEN_HEIGHT:
41             self.change_y *= -1
42
43
44 class MyGame(arcade.Window):
45     """ Our custom Window Class"""
46
47     def __init__(self):
48         """ Initializer """
49
50         # Call the parent class initializer
51         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
52
53         # Variables that will hold sprite lists
54         self.player_list = None
55         self.coin_list = None
56
57         # Set up the player info
58         self.player_sprite = None
59         self.score = 0
60
61         # Don't show the mouse cursor
62         self.set_mouse_visible(False)
63
64         arcade.set_background_color(arcade.color.AMAZON)
65
66     def setup(self):
67         """ Set up the game and initialize the variables. """

```

(continues on next page)

(continued from previous page)

```

68     # Sprite lists
69     self.player_list = arcade.SpriteList()
70     self.coin_list = arcade.SpriteList()
71
72     # Score
73     self.score = 0
74
75     # Set up the player
76     # Character image from kenney.nl
77     self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
78     self.player_sprite.center_x = 50
79     self.player_sprite.center_y = 50
80     self.player_list.append(self.player_sprite)
81
82     # Create the coins
83     for i in range(COIN_COUNT):
84
85         # Create the coin instance
86         # Coin image from kenney.nl
87         coin = Coin("coin_01.png", SPRITE_SCALING_COIN)
88
89         # Position the coin
90         coin.center_x = random.randrange(SCREEN_WIDTH)
91         coin.center_y = random.randrange(SCREEN_HEIGHT)
92         coin.change_x = random.randrange(-3, 4)
93         coin.change_y = random.randrange(-3, 4)
94
95         # Add the coin to the lists
96         self.coin_list.append(coin)
97
98     def on_draw(self):
99         """ Draw everything """
100        arcade.start_render()
101        self.coin_list.draw()
102        self.player_list.draw()
103
104        # Put the text on the screen.
105        output = f"Score: {self.score}"
106        arcade.draw_text(output, 10, 20, arcade.color.WHITE, 14)
107
108    def on_mouse_motion(self, x, y, dx, dy):
109        """ Handle Mouse Motion """
110
111        # Move the center of the player sprite to match the mouse x, y
112        self.player_sprite.center_x = x
113        self.player_sprite.center_y = y
114
115    def update(self, delta_time):
116        """ Movement and game logic """
117
118        # Call update on all sprites (The sprites don't do much in this
119        # example though.)
120        self.coin_list.update()
121
122        # Generate a list of all sprites that collided with the player.
123        hit_list = arcade.check_for_collision_with_list(self.player_sprite,
124                                                       self.coin_list)

```

(continues on next page)

(continued from previous page)

```

125
126     # Loop through each colliding sprite, remove it, and add to the score.
127     for coin in hit_list:
128         coin.kill()
129         self.score += 1
130
131
132 def main():
133     """ Main method """
134     window = MyGame()
135     window.setup()
136     arcade.run()
137
138
139 if __name__ == "__main__":
140     main()

```

TODO: Put in some text about spawning a sprite too close to the edge. Also make a refer to it from the final project.

Take what you've learned from the example above, and see if you can replicate this:

Fig. 36: Test Pattern

1.19.3 Coins Moving In Circles

Fig. 37: Coins moving in a circle

Listing 80: sprites_circle.py

```

1 import random
2 import arcade
3 import math
4
5 SPRITE_SCALING = 0.5
6
7 SCREEN_WIDTH = 800
8 SCREEN_HEIGHT = 600
9
10
11 class Coin(arcade.Sprite):
12
13     def __init__(self, filename, sprite_scaling):
14         """ Constructor. """
15         # Call the parent class (Sprite) constructor
16         super().__init__(filename, sprite_scaling)
17
18         # Current angle in radians
19         self.circle_angle = 0
20
21         # How far away from the center to orbit, in pixels
22         self.circle_radius = 0
23

```

(continues on next page)

(continued from previous page)

```

24     # How fast to orbit, in radians per frame
25     self.circle_speed = 0.008
26
27     # Set the center of the point we will orbit around
28     self.circle_center_x = 0
29     self.circle_center_y = 0
30
31 def update(self):
32
33     """ Update the ball's position. """
34     # Calculate a new x, y
35     self.center_x = self.circle_radius * math.sin(self.circle_angle) \
36         + self.circle_center_x
37     self.center_y = self.circle_radius * math.cos(self.circle_angle) \
38         + self.circle_center_y
39
40     # Increase the angle in prep for the next round.
41     self.circle_angle += self.circle_speed
42
43
44 class MyGame(arcade.Window):
45     """ Main application class. """
46
47     def __init__(self, width, height):
48
49         super().__init__(width, height)
50
51         # Sprite lists
52         self.player_list = None
53         self.coin_list = None
54
55         # Set up the player
56         self.score = 0
57         self.player_sprite = None
58
59     def start_new_game(self):
60         """ Set up the game and initialize the variables. """
61
62         # Sprite lists
63         self.player_list = arcade.SpriteList()
64         self.coin_list = arcade.SpriteList()
65
66         # Set up the player
67         self.score = 0
68
69         # Character image from kenney.nl
70         self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING)
71         self.player_sprite.center_x = 50
72         self.player_sprite.center_y = 70
73         self.player_list.append(self.player_sprite)
74
75     for i in range(50):
76
77         # Create the coin instance
78         # Coin image from kenney.nl
79         coin = Coin("coin_01.png", SPRITE_SCALING / 3)
80

```

(continues on next page)

(continued from previous page)

```

81     # Position the center of the circle the coin will orbit
82     coin.circle_center_x = random.randrange(SCREEN_WIDTH)
83     coin.circle_center_y = random.randrange(SCREEN_HEIGHT)
84
85     # Random radius from 10 to 200
86     coin.circle_radius = random.randrange(10, 200)
87
88     # Random start angle from 0 to 2pi
89     coin.circle_angle = random.random() * 2 * math.pi
90
91     # Add the coin to the lists
92     self.coin_list.append(coin)
93
94     # Don't show the mouse cursor
95     self.set_mouse_visible(False)
96
97     # Set the background color
98     arcade.set_background_color(arcade.color.AMAZON)
99
100    def on_draw(self):
101
102        # This command has to happen before we start drawing
103        arcade.start_render()
104
105        # Draw all the sprites.
106        self.coin_list.draw()
107        self.player_list.draw()
108
109        # Put the text on the screen.
110        output = "Score: " + str(self.score)
111        arcade.draw_text(output, 10, 20, arcade.color.WHITE, 14)
112
113    def on_mouse_motion(self, x, y, dx, dy):
114        self.player_sprite.center_x = x
115        self.player_sprite.center_y = y
116
117    def update(self, delta_time):
118        """ Movement and game logic """
119
120        # Call update on all sprites (The sprites don't do much in this
121        # example though.)
122        self.coin_list.update()
123
124        # Generate a list of all sprites that collided with the player.
125        hit_list = arcade.check_for_collision_with_list(self.player_sprite,
126                                                       self.coin_list)
127
128        # Loop through each colliding sprite, remove it, and add to the score.
129        for coin in hit_list:
130            self.score += 1
131            coin.kill()
132
133
134    def main():
135        window = MyGame(SCREEN_WIDTH, SCREEN_HEIGHT)
136        window.start_new_game()
137        arcade.run()

```

(continues on next page)

(continued from previous page)

```
138  
139  
140 if __name__ == "__main__":  
141     main()
```

Sprites can easily be rotated by setting their angle attribute. Angles are in degrees. So the following will flip the player upside down:

```
self.player_sprite.angle = 180
```

If you put this in the coin's update method, it would cause the coins to spin:

```
self.angle += 1  
  
# If we rotate past 360, reset it back a turn.  
if self.angle > 359:  
    self.angle -= 360
```

1.20 Debugging Programs

1.20.1 Understanding a Stack Trace

1.20.2 Use Print Statements

1.20.3 Google The Error

1.21 Using Sprites to Shoot

How do we get sprites that we can shoot?

Fig. 38: Coins shooting

1.21.1 Getting Started

First, let's go back to a program to start with.

Listing 81: Starting program for shooting sprites

```
1 import random  
2 import arcade  
3  
4 SPRITE_SCALING_PLAYER = 0.5  
5 SPRITE_SCALING_COIN = 0.2  
6 SPRITE_SCALING_LASER = 0.8  
7 COIN_COUNT = 50  
8  
9 SCREEN_WIDTH = 800  
10 SCREEN_HEIGHT = 600
```

(continues on next page)

(continued from previous page)

```

11
12 BULLET_SPEED = 5
13
14
15 class MyGame(arcade.Window):
16     """ Main application class. """
17
18     def __init__(self):
19         """ Initializer """
20
21         # Call the parent class initializer
22         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprites and Bullets Demo")
23
24         # Variables that will hold sprite lists
25         self.player_list = None
26         self.coin_list = None
27
28         # Set up the player info
29         self.player_sprite = None
30         self.score = 0
31
32         # Don't show the mouse cursor
33         self.set_mouse_visible(False)
34
35         arcade.set_background_color(arcade.color.AMAZON)
36
37     def setup(self):
38
39         """ Set up the game and initialize the variables. """
40
41         # Sprite lists
42         self.player_list = arcade.SpriteList()
43         self.coin_list = arcade.SpriteList()
44
45         # Set up the player
46         self.score = 0
47
48         # Image from kenney.nl
49         self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
50         self.player_sprite.center_x = 50
51         self.player_sprite.center_y = 70
52         self.player_list.append(self.player_sprite)
53
54         # Create the coins
55         for i in range(COIN_COUNT):
56
57             # Create the coin instance
58             # Coin image from kenney.nl
59             coin = arcade.Sprite("coin_01.png", SPRITE_SCALING_COIN)
60
61             # Position the coin
62             coin.center_x = random.randrange(SCREEN_WIDTH)
63             coin.center_y = random.randrange(SCREEN_HEIGHT)
64
65             # Add the coin to the lists
66             self.coin_list.append(coin)
67
68         # Set the background color

```

(continues on next page)

(continued from previous page)

```

68     arcade.set_background_color(arcade.color.AMAZON)
69
70     def on_draw(self):
71         """
72             Render the screen.
73         """
74
75         # This command has to happen before we start drawing
76         arcade.start_render()
77
78         # Draw all the sprites.
79         self.coin_list.draw()
80         self.player_list.draw()
81
82         # Render the text
83         arcade.draw_text(f"Score: {self.score}", 10, 20, arcade.color.WHITE, 14)
84
85     def on_mouse_motion(self, x, y, dx, dy):
86         """
87             Called whenever the mouse moves.
88         """
89         self.player_sprite.center_x = x
90         self.player_sprite.center_y = y
91
92     def on_mouse_press(self, x, y, button, modifiers):
93         """
94             Called whenever the mouse button is clicked.
95         """
96         pass
97
98     def update(self, delta_time):
99         """
100            Movement and game logic
101        """
102
103        # Call update on all sprites
104        self.coin_list.update()
105
106    def main():
107        window = MyGame()
108        window.setup()
109        arcade.run()
110
111 if __name__ == "__main__":
112     main()

```

If you run this program, the player should move around the screen, and their should be coins. But not much else is happening yet.

Next, we need a ‘shooting’ image:



Fig. 39: laserBlue01.png

Download this image (originally from Kenney.nl) and make sure it is in the same folder as your code.

1.21.2 Keeping The Player At The Bottom

Right now the player can move anywhere on the screen. We want to keep that sprite fixed to the bottom of the screen. To do that, just remove the line of code for moving the player on the y-axis. The player will keep the same y value that we set back in the `setup` method.

```
def on_mouse_motion(self, x, y, dx, dy):
    """
    Called whenever the mouse moves.
    """

    self.player_sprite.center_x = x
    # REMOVE THIS LINE: self.player_sprite.center_y = y
```

1.21.3 Moving The Coins Up

We want all the coins above the player. So we can adjust the starting y locations to have a starting point of 150 instead of 0. That will keep them above the player.

```
# Create the coins
for i in range(COIN_COUNT):

    # Create the coin instance
    # Coin image from kenney.nl
    coin = arcade.Sprite("coin_01.png", SPRITE_SCALING_COIN)

    # Position the coin
    coin.center_x = random.randrange(SCREEN_WIDTH)
    coin.center_y = random.randrange(150, SCREEN_HEIGHT)

    # Add the coin to the lists
    self.coin_list.append(coin)
```

1.21.4 Set Up Bullet List

We need to create a list to manage the bullets. There are four places we need to add this `bullet_list` code:

- Create the `bullet_list` variable (Line 26)
- Create an instance of `SpriteList` (Line 44)
- Draw the bullet list (Line 83)
- Update the bullet list (Line 105)

Listing 82: Set up bullet list

```
1 import random
2 import arcade
3
4 SPRITE_SCALING_PLAYER = 0.5
5 SPRITE_SCALING_COIN = 0.2
6 SPRITE_SCALING_LASER = 0.8
7 COIN_COUNT = 50
8
9 SCREEN_WIDTH = 800
```

(continues on next page)

(continued from previous page)

```

10 SCREEN_HEIGHT = 600
11
12 BULLET_SPEED = 5
13
14
15 class MyGame(arcade.Window):
16     """ Main application class. """
17
18     def __init__(self):
19         """ Initializer """
20
21         # Call the parent class initializer
22         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprites and Bullets Demo")
23
24         # Variables that will hold sprite lists
25         self.player_list = None
26         self.coin_list = None
27         self.bullet_list = None
28
29         # Set up the player info
30         self.player_sprite = None
31         self.score = 0
32
33         # Don't show the mouse cursor
34         self.set_mouse_visible(False)
35
36         arcade.set_background_color(arcade.color.AMAZON)
37
38     def setup(self):
39
40         """ Set up the game and initialize the variables. """
41
42         # Sprite lists
43         self.player_list = arcade.SpriteList()
44         self.coin_list = arcade.SpriteList()
45         self.bullet_list = arcade.SpriteList()
46
47         # Set up the player
48         self.score = 0
49
50         # Image from kenney.nl
51         self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
52         self.player_sprite.center_x = 50
53         self.player_sprite.center_y = 70
54         self.player_list.append(self.player_sprite)
55
56         # Create the coins
57         for i in range(COIN_COUNT):
58
59             # Create the coin instance
60             # Coin image from kenney.nl
61             coin = arcade.Sprite("coin_01.png", SPRITE_SCALING_COIN)
62
63             # Position the coin
64             coin.center_x = random.randrange(SCREEN_WIDTH)
65             coin.center_y = random.randrange(150, SCREEN_HEIGHT)
66
67             # Add the coin to the lists

```

(continues on next page)

(continued from previous page)

```

67         self.coin_list.append(coin)

68     # Set the background color
69     arcade.set_background_color(arcade.color.AMAZON)

70
71     def on_draw(self):
72         """
73             Render the screen.
74         """
75
76
77         # This command has to happen before we start drawing
78         arcade.start_render()

79
80         # Draw all the sprites.
81         self.coin_list.draw()
82         self.player_list.draw()
83         self.bullet_list.draw()

84
85         # Render the text
86         arcade.draw_text(f"Score: {self.score}", 10, 20, arcade.color.WHITE, 14)

87
88     def on_mouse_motion(self, x, y, dx, dy):
89         """
90             Called whenever the mouse moves.
91         """
92         self.player_sprite.center_x = x

93
94     def on_mouse_press(self, x, y, button, modifiers):
95         """
96             Called whenever the mouse button is clicked.
97         """
98         pass

99
100    def update(self, delta_time):
101        """
102            Movement and game logic """
103
104        # Call update on all sprites
105        self.coin_list.update()
106        self.bullet_list.update()

107
108    def main():
109        window = MyGame()
110        window.setup()
111        arcade.run()

112
113
114    if __name__ == "__main__":
115        main()

```

1.21.5 Creating Bullets

We need to create bullets when the user presses the mouse button. We can add an `on_mouse_press` method to do something when the user presses the mouse button:

```
def on_mouse_press(self, x, y, button, modifiers):  
  
    # Create a bullet  
    bullet = arcade.Sprite("laserBlue01.png", SPRITE_SCALING_LASER)  
  
    # Add the bullet to the appropriate list  
    self.bullet_list.append(bullet)
```

This will create a bullet, but the bullet will default to the lower left corner. You can just barely see it.

We can give the bullet a position:

```
def on_mouse_press(self, x, y, button, modifiers):  
  
    # Create a bullet  
    bullet = arcade.Sprite("laserBlue01.png", SPRITE_SCALING_LASER)  
  
    bullet.center_x = x  
    bullet.center_y = y  
  
    # Add the bullet to the appropriate list  
    self.bullet_list.append(bullet)
```

But this isn't what we want either. The code above puts the laser where we click the mouse. We want the laser to be where the player is. That's easy:

```
def on_mouse_press(self, x, y, button, modifiers):  
  
    # Create a bullet  
    bullet = arcade.Sprite("laserBlue01.png", SPRITE_SCALING_LASER)  
  
    bullet.center_x = self.player_sprite.center_x  
    bullet.center_y = self.player_sprite.center_y  
  
    # Add the bullet to the appropriate list  
    self.bullet_list.append(bullet)
```

We can even start the player a bit ABOVE the player:

```
def on_mouse_press(self, x, y, button, modifiers):  
  
    # Create a bullet  
    bullet = arcade.Sprite("laserBlue01.png", SPRITE_SCALING_LASER)  
  
    bullet.center_x = self.player_sprite.center_x  
    bullet.center_y = self.player_sprite.center_y + 30  
  
    # Add the bullet to the appropriate list  
    self.bullet_list.append(bullet)
```

We can make the bullet move up using the constant BULLET_SPEED which we set to 5 at the top of the program:

```
def on_mouse_press(self, x, y, button, modifiers):  
  
    # Create a bullet  
    bullet = arcade.Sprite("laserBlue01.png", SPRITE_SCALING_LASER)
```

(continues on next page)

(continued from previous page)

```

bullet.center_x = self.player_sprite.center_x
bullet.center_y = self.player_sprite.center_y + 30
bullet.change_y = BULLET_SPEED

# Add the bullet to the appropriate list
self.bullet_list.append(bullet)

```

We can rotate the bullet so it isn't sideways using the angle attribute built into the Sprite class:

```

def on_mouse_press(self, x, y, button, modifiers):

    # Create a bullet
    bullet = arcade.Sprite("laserBlue01.png", SPRITE_SCALING_LASER)

    bullet.center_x = self.player_sprite.center_x
    bullet.center_y = self.player_sprite.center_y + 30
    bullet.change_y = BULLET_SPEED
    bullet.angle = 90

    # Add the bullet to the appropriate list
    self.bullet_list.append(bullet)

```

1.21.6 Bullet Collisions

Now that we have bullets, how do we get them to collide with the coins? We add the following to our applications update method:

```

# Loop through each bullet
for bullet in self.bullet_list:

    # Check this bullet to see if it hit a coin
    hit_list = arcade.check_for_collision_with_list(bullet, self.coin_list)

    # If it did, get rid of the bullet
    if len(hit_list) > 0:
        bullet.kill()

    # For every coin we hit, add to the score and remove the coin
    for coin in hit_list:
        coin.kill()
        self.score += 1

    # If the bullet flies off-screen, remove it.
    if bullet.bottom > SCREEN_HEIGHT:
        bullet.kill()

```

We loop through each bullet with a `for` loop. Then we check to see if the bullet is hitting any of the coins. If it is, we get rid of the coin. We get rid of the bullet.

We also check to see if the bullet flies off the top of the screen. If it does, we get rid of the bullet. This is easy to forget, but if you do, it will cause the computer to slow down because you are tracking thousands of bullets that have long ago left the space we care about.

Here's the full example:

Listing 83: sprites_bullet.py

```
1 import random
2 import arcade
3
4 SPRITE_SCALING_PLAYER = 0.5
5 SPRITE_SCALING_COIN = 0.2
6 SPRITE_SCALING_LASER = 0.8
7 COIN_COUNT = 50
8
9 SCREEN_WIDTH = 800
10 SCREEN_HEIGHT = 600
11
12 BULLET_SPEED = 5
13
14
15 class MyGame(arcade.Window):
16     """ Main application class. """
17
18     def __init__(self):
19         """Initializer"""
20         # Call the parent class initializer
21         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprites and Bullets Demo")
22
23         # Variables that will hold sprite lists
24         self.player_list = None
25         self.coin_list = None
26         self.bullet_list = None
27
28         # Set up the player info
29         self.player_sprite = None
30         self.score = 0
31
32         # Don't show the mouse cursor
33         self.set_mouse_visible(False)
34
35         arcade.set_background_color(arcade.color.AMAZON)
36
37     def setup(self):
38
39         """ Set up the game and initialize the variables. """
40
41         # Sprite lists
42         self.player_list = arcade.SpriteList()
43         self.coin_list = arcade.SpriteList()
44         self.bullet_list = arcade.SpriteList()
45
46         # Set up the player
47         self.score = 0
48
49         # Image from kenney.nl
50         self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING_PLAYER)
51         self.player_sprite.center_x = 50
52         self.player_sprite.center_y = 70
53         self.player_list.append(self.player_sprite)
54
55         # Create the coins
```

(continues on next page)

(continued from previous page)

```

56     for i in range(COIN_COUNT):
57
58         # Create the coin instance
59         # Coin image from kenney.nl
60         coin = arcade.Sprite("coin_01.png", SPRITE_SCALING_COIN)
61
62         # Position the coin
63         coin.center_x = random.randrange(SCREEN_WIDTH)
64         coin.center_y = random.randrange(120, SCREEN_HEIGHT)
65
66         # Add the coin to the lists
67         self.coin_list.append(coin)
68
69         # Set the background color
70         arcade.set_background_color(arcade.color.AMAZON)
71
72     def on_draw(self):
73         """
74             Render the screen.
75         """
76
77         # This command has to happen before we start drawing
78         arcade.start_render()
79
80         # Draw all the sprites.
81         self.coin_list.draw()
82         self.bullet_list.draw()
83         self.player_list.draw()
84
85         # Render the text
86         arcade.draw_text(f"Score: {self.score}", 10, 20, arcade.color.WHITE, 14)
87
88     def on_mouse_motion(self, x, y, dx, dy):
89         """
90             Called whenever the mouse moves.
91         """
92         self.player_sprite.center_x = x
93
94     def on_mouse_press(self, x, y, button, modifiers):
95         """
96             Called whenever the mouse button is clicked.
97         """
98
99         # Create a bullet
100        bullet = arcade.Sprite("laserBlue01.png", SPRITE_SCALING_LASER)
101
102        # The image points to the right, and we want it to point up. So
103        # rotate it.
104        bullet.angle = 90
105
106        # Position the bullet
107        bullet.center_x = self.player_sprite.center_x
108        bullet.bottom = self.player_sprite.top
109
110        # Add the bullet to the appropriate lists
111        self.bullet_list.append(bullet)
112

```

(continues on next page)

(continued from previous page)

```
113     def update(self, delta_time):
114         """ Movement and game logic """
115
116         # Call update on all sprites
117         self.coin_list.update()
118         self.bullet_list.update()
119
120         # Loop through each bullet
121         for bullet in self.bullet_list:
122
123             # Check this bullet to see if it hit a coin
124             hit_list = arcade.check_for_collision_with_list(bullet, self.coin_list)
125
126             # If it did, get rid of the bullet
127             if len(hit_list) > 0:
128                 bullet.kill()
129
130             # For every coin we hit, add to the score and remove the coin
131             for coin in hit_list:
132                 coin.kill()
133                 self.score += 1
134
135             # If the bullet flies off-screen, remove it.
136             if bullet.bottom > SCREEN_HEIGHT:
137                 bullet.kill()
138
139
140     def main():
141         window = MyGame()
142         window.setup()
143         arcade.run()
144
145
146     if __name__ == "__main__":
147         main()
```

1.21.7 Other Examples

Here are some other examples from Arcade Academy and its [Example List](#):

- Aim bullets with mouse
- Enemies shoot periodically
- Enemies shoot at random times
- Enemies aim at player
- Explosions

1.22 Sprites and Walls

1.22.1 Setup

Many games with sprites often have “walls” that the character can’t move through. There are rather straight-forward to create.

To begin with, let’s get a couple images. Our character, and a box that will act as a blocking wall. Both images are from kenney.nl.



Fig. 40: images/character.png



Fig. 41: images/boxCrate_double.png

Start with a default file:

Listing 84: sprite_move_walls.py start

```

1  """ Sprite Sample Program """
2
3  import arcade
4
5  # --- Constants ---
6  SPRITE_SCALING_BOX = 0.5
7  SPRITE_SCALING_PLAYER = 0.5
8
9  SCREEN_WIDTH = 800
10 SCREEN_HEIGHT = 600
11
12 MOVEMENT_SPEED = 5
13
14
15 class MyGame(arcade.Window):
16     """ This class represents the main window of the game. """
17
18     def __init__(self):
19         """ Initializer """
20         # Call the parent class initializer
21         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprites With Walls Example")
22
23
24     def setup(self):
25         # Set the background color
26         arcade.set_background_color(arcade.color.AMAZON)
27

```

(continues on next page)

(continued from previous page)

```
28     def on_draw(self):
29         arcade.start_render()
30
31
32     def main():
33         window = MyGame()
34         window.setup()
35         arcade.run()
36
37
38 if __name__ == "__main__":
39     main()
```

In the `__init__` method, let's create some variables to hold our sprites:

```
# Sprite lists
self.player_list = None
self.wall_list = None

# Set up the player
self.player_sprite = None

# This variable holds our simple "physics engine"
self.physics_engine = None
```

In the `setup` method, let's create our sprite lists:

```
# Sprite lists
self.player_list = arcade.SpriteList()
self.wall_list = arcade.SpriteList()
```

Then reset the score and create the player:

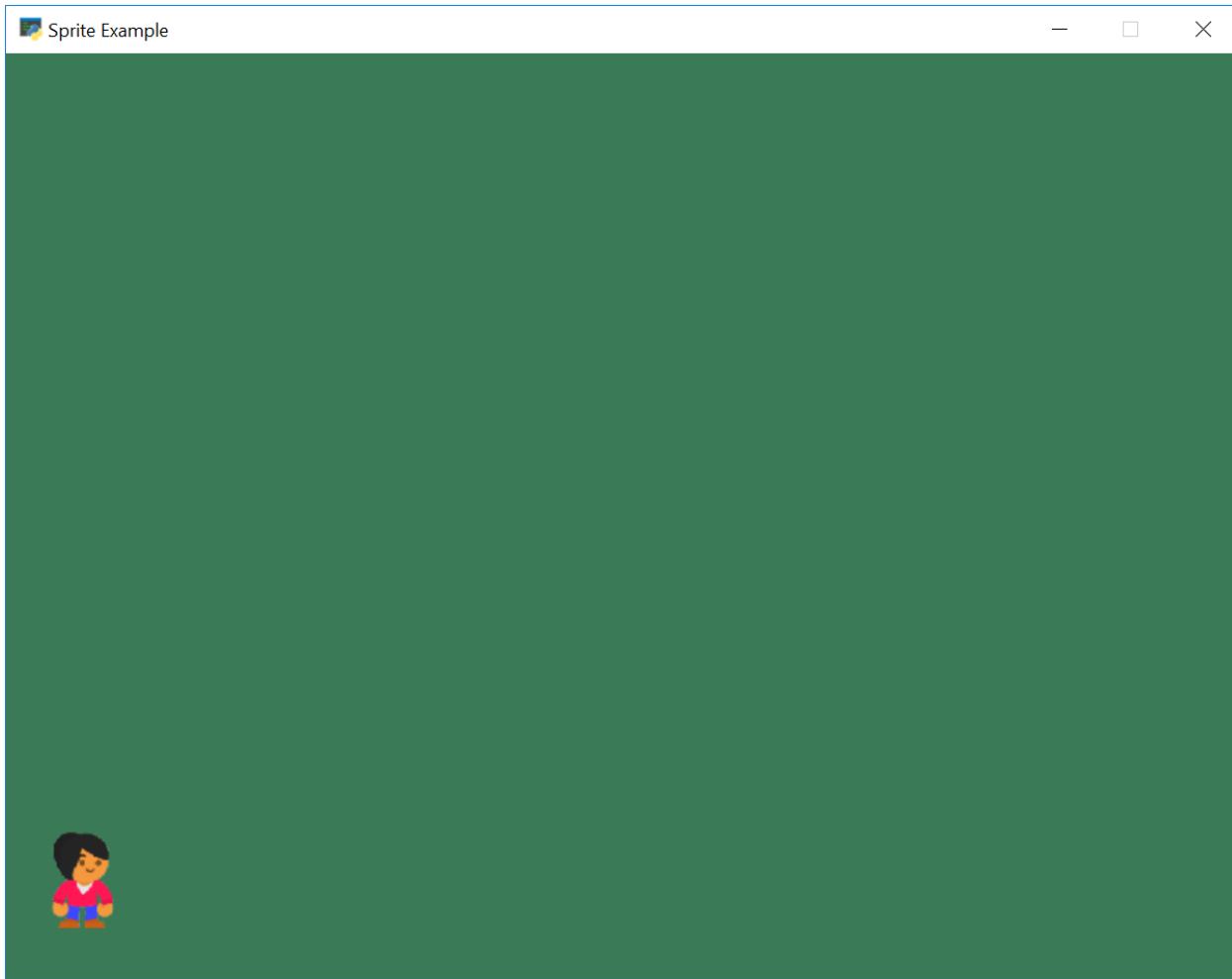
```
# Reset the score
self.score = 0

# Create the player
self.player_sprite = arcade.Sprite("images/character.png", SPRITE_SCALING_PLAYER)
self.player_sprite.center_x = 50
self.player_sprite.center_y = 64
self.player_list.append(self.player_sprite)
```

Then go ahead and draw everything in our `on_draw`:

```
def on_draw(self):
    arcade.start_render()
    self.wall_list.draw()
    self.player_list.draw()
```

Run the program and make sure it works.



1.22.2 Individually Placing Walls

In our `setup` method, we can position individual boxes to be used as “walls”:

```
# Manually create and position a box at 300, 200
wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
wall.center_x = 300
wall.center_y = 200
self.wall_list.append(wall)

# Manually create and position a box at 364, 200
wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
wall.center_x = 364
wall.center_y = 200
self.wall_list.append(wall)
```

Go ahead and try it out. It should look like:



Full listing below:

Listing 85: sprite_move_walls.py Step 2

```
1  """ Sprite Sample Program """
2
3  import arcade
4
5  # --- Constants ---
6  SPRITE_SCALING_BOX = 0.5
7  SPRITE_SCALING_PLAYER = 0.5
8
9  SCREEN_WIDTH = 800
10 SCREEN_HEIGHT = 600
11
12 MOVEMENT_SPEED = 5
13
14
15 class MyGame(arcade.Window):
16     """ This class represents the main window of the game. """
17
18     def __init__(self):
19         """ Initializer """
20         # Call the parent class initializer
```

(continues on next page)

(continued from previous page)

```

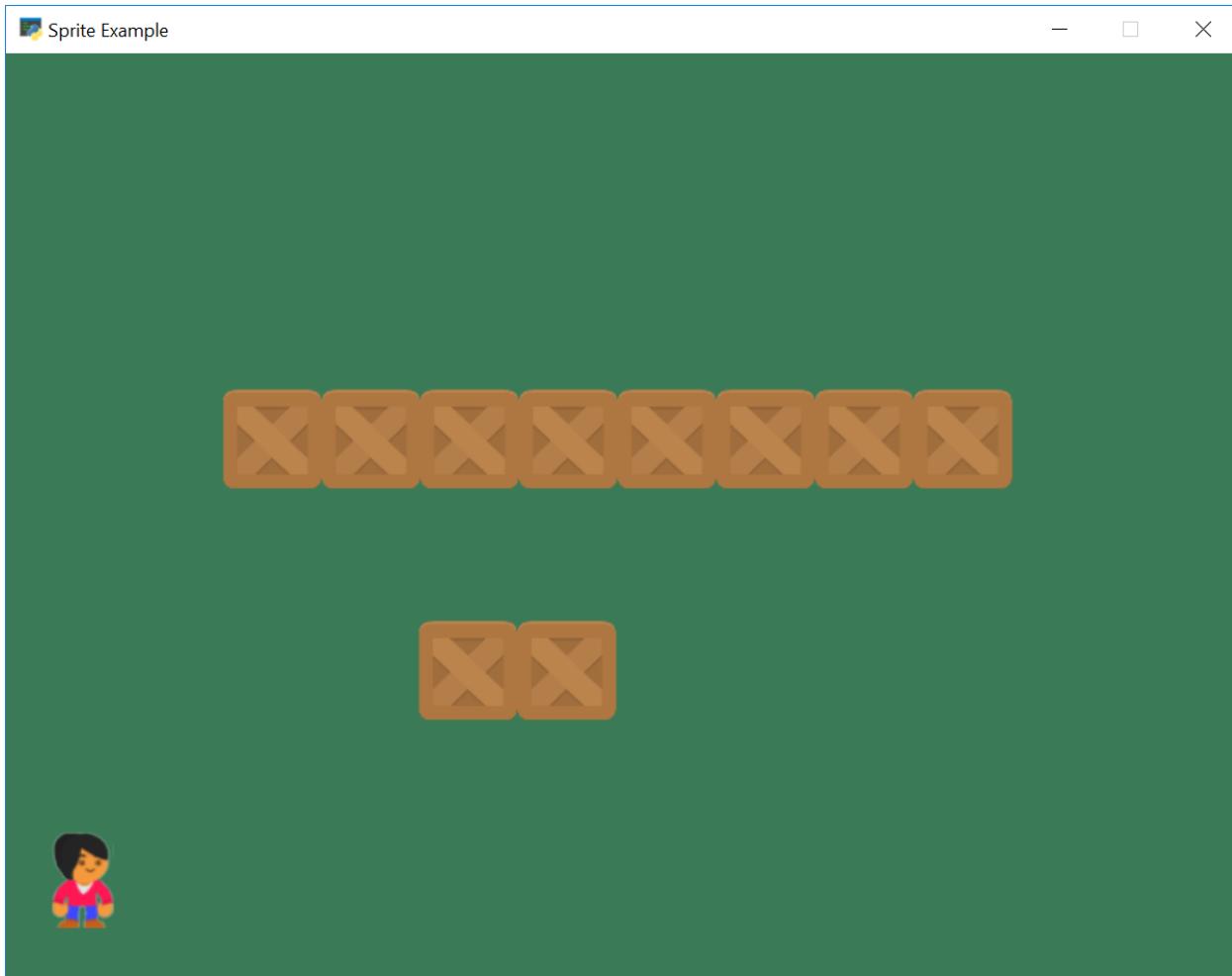
21     super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
22
23     # Sprite lists
24     self.player_list = None
25     self.wall_list = None
26
27     # Set up the player
28     self.player_sprite = None
29
30     def setup(self):
31
32         # Set the background color
33         arcade.set_background_color(arcade.color.AMAZON)
34
35         # Sprite lists
36         self.player_list = arcade.SpriteList()
37         self.wall_list = arcade.SpriteList()
38
39         # Reset the score
40         self.score = 0
41
42         # Create the player
43         self.player_sprite = arcade.Sprite("images/character.png", SPRITE_SCALING_
44             ↪PLAYER)
44         self.player_sprite.center_x = 50
45         self.player_sprite.center_y = 64
46         self.player_list.append(self.player_sprite)
47
48         # Manually create and position a box at 300, 200
49         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
50         wall.center_x = 300
51         wall.center_y = 200
52         self.wall_list.append(wall)
53
54         # Manually create and position a box at 364, 200
55         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
56         wall.center_x = 364
57         wall.center_y = 200
58         self.wall_list.append(wall)
59
60     def on_draw(self):
61         arcade.start_render()
62         self.wall_list.draw()
63         self.player_list.draw()
64
65
66     def main():
67         """ Main method """
68         window = MyGame()
69         window.setup()
70         arcade.run()
71
72
73     if __name__ == "__main__":
74         main()

```

1.22.3 Placing Walls With A Loop

In our setup method, we can create a row of box sprites using a `for` loop. In the code below, our `y` value is always 350, and we change the `x` value from 173 to 650. We put a box every 64 pixels because each box happens to be 64 pixels wide.

```
# Place boxes inside a loop
for x in range(173, 650, 64):
    wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
    wall.center_x = x
    wall.center_y = 350
    self.wall_list.append(wall)
```



1.22.4 Placing Walls With A List

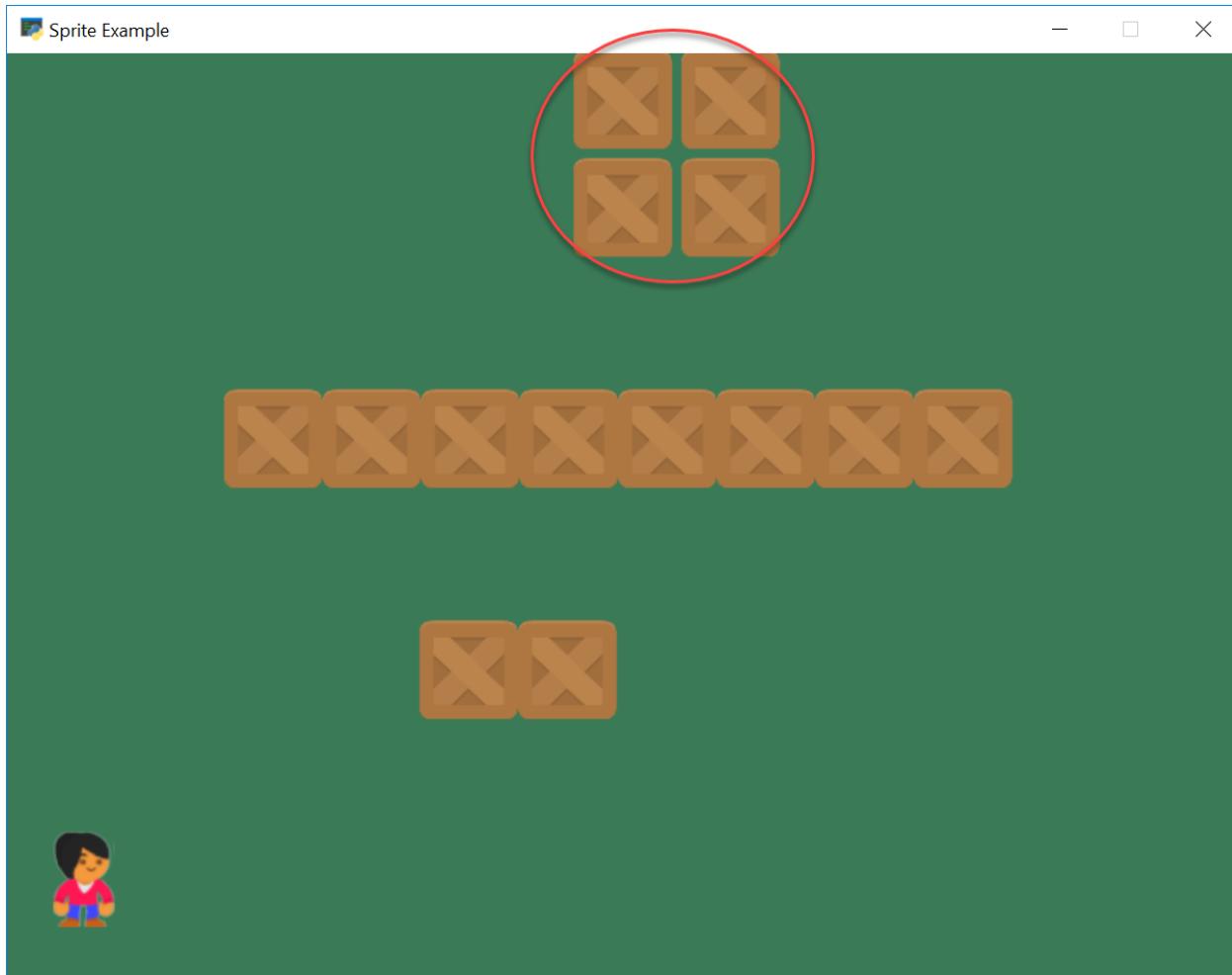
You could even create a list of coordinates, and then just loop through that list creating walls:

```
# --- Place walls with a list
coordinate_list = [[400, 500],
                  [470, 500],
                  [400, 570],
                  [470, 570]]
```

(continues on next page)

(continued from previous page)

```
# Loop through coordinates
for coordinate in coordinate_list:
    wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
    wall.center_x = coordinate[0]
    wall.center_y = coordinate[1]
    self.wall_list.append(wall)
```



Full listing below:

Listing 86: sprite_move_walls.py Step 3

```
1  """ Sprite Sample Program """
2
3  import arcade
4
5  # --- Constants ---
6  SPRITE_SCALING_BOX = 0.5
7  SPRITE_SCALING_PLAYER = 0.5
8
9  SCREEN_WIDTH = 800
10 SCREEN_HEIGHT = 600
```

(continues on next page)

(continued from previous page)

```

11 MOVEMENT_SPEED = 5
12
13
14 class MyGame(arcade.Window):
15     """ This class represents the main window of the game. """
16
17     def __init__(self):
18         """ Initializer """
19         # Call the parent class initializer
20         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
21
22         # Sprite lists
23         self.player_list = None
24         self.wall_list = None
25
26         # Set up the player
27         self.player_sprite = None
28
29     def setup(self):
30
31         # Set the background color
32         arcade.set_background_color(arcade.color.AMAZON)
33
34         # Sprite lists
35         self.player_list = arcade.SpriteList()
36         self.wall_list = arcade.SpriteList()
37
38         # Reset the score
39         self.score = 0
40
41         # Create the player
42         self.player_sprite = arcade.Sprite("images/character.png", SPRITE_SCALING_
43         ↪PLAYER)
44         self.player_sprite.center_x = 50
45         self.player_sprite.center_y = 64
46         self.player_list.append(self.player_sprite)
47
48         # --- Manually place walls
49
50         # Manually create and position a box at 300, 200
51         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
52         wall.center_x = 300
53         wall.center_y = 200
54         self.wall_list.append(wall)
55
56         # Manually create and position a box at 364, 200
57         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
58         wall.center_x = 364
59         wall.center_y = 200
60         self.wall_list.append(wall)
61
62         # --- Place boxes inside a loop
63         for x in range(173, 650, 64):
64             wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
65             wall.center_x = x
66             wall.center_y = 350
67             self.wall_list.append(wall)

```

(continues on next page)

(continued from previous page)

```

67
68     # --- Place walls with a list
69     coordinate_list = [[400, 500],
70                         [470, 500],
71                         [400, 570],
72                         [470, 570]]
73
74     # Loop through coordinates
75     for coordinate in coordinate_list:
76         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
77         wall.center_x = coordinate[0]
78         wall.center_y = coordinate[1]
79         self.wall_list.append(wall)
80
81
82     def on_draw(self):
83         arcade.start_render()
84         self.player_list.draw()
85         self.wall_list.draw()
86
87
88 def main():
89     """ Main method """
90     window = MyGame()
91     window.setup()
92     arcade.run()
93
94
95 if __name__ == "__main__":
96     main()

```

1.22.5 Physics Engine

First, we need to hook the keyboard up to the player:

```

def on_key_press(self, key, modifiers):
    """Called whenever a key is pressed. """

    if key == arcade.key.UP:
        self.player_sprite.change_y = MOVEMENT_SPEED
    elif key == arcade.key.DOWN:
        self.player_sprite.change_y = -MOVEMENT_SPEED
    elif key == arcade.key.LEFT:
        self.player_sprite.change_x = -MOVEMENT_SPEED
    elif key == arcade.key.RIGHT:
        self.player_sprite.change_x = MOVEMENT_SPEED

def on_key_release(self, key, modifiers):
    """Called when the user releases a key. """

    if key == arcade.key.UP or key == arcade.key.DOWN:
        self.player_sprite.change_y = 0
    elif key == arcade.key.LEFT or key == arcade.key.RIGHT:
        self.player_sprite.change_x = 0

```

Now, we need to add a way to stop the player from running into walls.

The Arcade Library has a built in “physics engine.” A physics engine handles the interactions between the virtual physical objects in the game. For example, a physics engine might be several balls running into each other, a character sliding down a hill, or a car making a turn on the road.

Physics engines have made impressive gains on what they can simulate. For our game, we’ll just keep things simple and make sure our character can’t walk through walls.

We’ll create variable to hold our physics engine in the `__init__`:

```
# This variable holds our simple "physics engine"
self.physics_engine = None
```

We can create the actual physics engine in our `setup` method with the following code:

```
self.physics_engine = arcade.PhysicsEngineSimple(self.player_sprite, self.wall_list)
```

This identifies the player character (`player_sprite`), and a list of sprites (`wall_list`) that the player character isn’t allowed to pass through.

Before, we updated all the sprites with a `self.all_sprites_list.update()` command. With the physics engine, we will instead update the sprites by using the physics engine’s `update`:

```
def update(self, delta_time):
    self.physics_engine.update()
```

The simple physics engine follows the following algorithm:

- Move the player in the x direction according to the player’s `change_x` value.
- Check the player against the wall list and see if there are any collisions.
- If the player is colliding:
 - If the player is moving right, set the player’s right edge to the wall’s left edge.
 - If the player is moving left, set the player’s left edge to the wall’s right edge.
 - If the player isn’t moving left or right, print out a message that we are confused how we hit something when we aren’t moving.
- Then we just do the same thing, except with the y coordinates.

You can see the [physics engine source code](#) on GitHub.

Here’s the full example:

Listing 87: `sprite_move_walls.py`

```
1 """ Sprite Sample Program """
2
3 import arcade
4
5 # --- Constants ---
6 SPRITE_SCALING_BOX = 0.5
7 SPRITE_SCALING_PLAYER = 0.5
8
9 SCREEN_WIDTH = 800
10 SCREEN_HEIGHT = 600
11
12 MOVEMENT_SPEED = 5
13
14
```

(continues on next page)

(continued from previous page)

```

15 class MyGame(arcade.Window):
16     """ This class represents the main window of the game. """
17
18     def __init__(self):
19         """ Initializer """
20         # Call the parent class initializer
21         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
22
23         # Sprite lists
24         self.player_list = None
25         self.wall_list = None
26
27         # Set up the player
28         self.player_sprite = None
29
30         # This variable holds our simple "physics engine"
31         self.physics_engine = None
32
33     def setup(self):
34
35         # Set the background color
36         arcade.set_background_color(arcade.color.AMAZON)
37
38         # Sprite lists
39         self.player_list = arcade.SpriteList()
40         self.wall_list = arcade.SpriteList()
41
42         # Reset the score
43         self.score = 0
44
45         # Create the player
46         self.player_sprite = arcade.Sprite("images/character.png", SPRITE_SCALING_
47             ↪PLAYER)
47         self.player_sprite.center_x = 50
48         self.player_sprite.center_y = 64
49         self.player_list.append(self.player_sprite)
50
51         # --- Manually place walls
52
53         # Manually create and position a box at 300, 200
54         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
55         wall.center_x = 300
56         wall.center_y = 200
57         self.wall_list.append(wall)
58
59         # Manually create and position a box at 364, 200
60         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
61         wall.center_x = 364
62         wall.center_y = 200
63         self.wall_list.append(wall)
64
65         # --- Place boxes inside a loop
66         for x in range(173, 650, 64):
67             wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
68             wall.center_x = x
69             wall.center_y = 350
70             self.wall_list.append(wall)

```

(continues on next page)

(continued from previous page)

```

71
72     # --- Place walls with a list
73     coordinate_list = [[400, 500],
74                         [470, 500],
75                         [400, 570],
76                         [470, 570]]
77
78     # Loop through coordinates
79     for coordinate in coordinate_list:
80         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
81         wall.center_x = coordinate[0]
82         wall.center_y = coordinate[1]
83         self.wall_list.append(wall)
84
85     # Create the physics engine. Give it a reference to the player, and
86     # the walls we can't run into.
87     self.physics_engine = arcade.PhysicsEngineSimple(self.player_sprite, self.
88     ↪wall_list)
89
90     def on_draw(self):
91         arcade.start_render()
92         self.wall_list.draw()
93         self.player_list.draw()
94
95     def update(self, delta_time):
96         self.physics_engine.update()
97
98     def on_key_press(self, key, modifiers):
99         """Called whenever a key is pressed. """
100
101         if key == arcade.key.UP:
102             self.player_sprite.change_y = MOVEMENT_SPEED
103         elif key == arcade.key.DOWN:
104             self.player_sprite.change_y = -MOVEMENT_SPEED
105         elif key == arcade.key.LEFT:
106             self.player_sprite.change_x = -MOVEMENT_SPEED
107         elif key == arcade.key.RIGHT:
108             self.player_sprite.change_x = MOVEMENT_SPEED
109
110     def on_key_release(self, key, modifiers):
111         """Called when the user releases a key. """
112
113         if key == arcade.key.UP or key == arcade.key.DOWN:
114             self.player_sprite.change_y = 0
115         elif key == arcade.key.LEFT or key == arcade.key.RIGHT:
116             self.player_sprite.change_x = 0
117
118     def main():
119         """ Main method """
120         window = MyGame()
121         window.setup()
122         arcade.run()
123
124
125     if __name__ == "__main__":
126         main()

```

1.22.6 Using a View Port for Scrolling

What if one screen isn't enough to hold your maze of walls? We can have a world that is larger than just our window. We do this by adjusting the *view port*. Normally coordinate (0, 0) is the lower left corner of our screen. We can change that! We could have an entire world stretch from (0, 0) to (3000, 3000), and have a smaller view port that was 800x640 that scrolled around that.

The command for using the view port is `set_viewport`. This command takes four arguments. The first two are the left and bottom boundaries of the window. By default these are zero. That is why (0, 0) is in the lower left of the screen. The next two commands are the top and right coordinates of the screen. By default these are the screen width and height, minus one. So an 800 pixel-wide window would have x-coordinates from 0 - 799.

A command like this would shift the whole “view” of the window 200 pixels to the right:

```
# Specify viewport size by (left, right, bottom, top)
arcade.set_viewport(200, 200 + SCREEN_WIDTH - 1, 0, SCREEN_HEIGHT - 1)
```

So with a 800 wide pixel window, we would show x-coordinates 200 - 999 instead of 0 - 799.

Instead of hard-coding the shift at 200 pixels, we need to use a variable and have rules around when to shift the view. In our next example, we will create two new instance variables in our application class that hold the left and bottom coordinates for our view port. We'll default to zero.

```
self.view_left = 0
self.view_bottom = 0
```

We are also going to create two new constants. We don't want the player to reach the edge of the screen before we start scrolling. Because then the player would have no idea where she is going. In our example we will set a “margin” of 40 pixels. When the player is 40 pixels from the edge of the screen, we'll move the view port so she can see at least 40 pixels around her.

```
VIEWPORT_MARGIN = 40
```

Next, in our update method, we need to see if the user has moved too close to the edge of the screen and we need to update the boundaries.

```
# Keep track of if we changed the boundary. We don't want to call the
# set_viewport command if we didn't change the view port.
changed = False

# Scroll left
left_bndry = self.view_left + VIEWPORT_MARGIN
if self.player_sprite.left < left_bndry:
    self.view_left -= left_bndry - self.player_sprite.left
    changed = True

# Scroll right
right_bndry = self.view_left + SCREEN_WIDTH - VIEWPORT_MARGIN
if self.player_sprite.right > right_bndry:
    self.view_left += self.player_sprite.right - right_bndry
    changed = True

# Scroll up
top_bndry = self.view_bottom + SCREEN_HEIGHT - VIEWPORT_MARGIN
if self.player_sprite.top > top_bndry:
    self.view_bottom += self.player_sprite.top - top_bndry
    changed = True
```

(continues on next page)

(continued from previous page)

```

# Scroll down
bottom_bndry = self.view_bottom + VIEWPORT_MARGIN
if self.player_sprite.bottom < bottom_bndry:
    self.view_bottom -= bottom_bndry - self.player_sprite.bottom
    changed = True

# Make sure our boundaries are integer values. While the view port does
# support floating point numbers, for this application we want every pixel
# in the view port to map directly onto a pixel on the screen. We don't want
# any rounding errors.
self.view_left = int(self.view_left)
self.view_bottom = int(self.view_bottom)

# If we changed the boundary values, update the view port to match
if changed:
    arcade.set_viewport(self.view_left,
                        SCREEN_WIDTH + self.view_left - 1,
                        self.view_bottom,
                        SCREEN_HEIGHT + self.view_bottom - 1)

```

The full example is below:

Listing 88: sprite_move_scrolling.py

```

1 """ Sprite Sample Program """
2
3 import arcade
4
5 # --- Constants ---
6 SPRITE_SCALING_BOX = 0.5
7 SPRITE_SCALING_PLAYER = 0.5
8
9 SCREEN_WIDTH = 800
10 SCREEN_HEIGHT = 600
11
12 MOVEMENT_SPEED = 5
13
14 VIEWPORT_MARGIN = 40
15
16
17 class MyGame(arcade.Window):
18     """ This class represents the main window of the game. """
19
20     def __init__(self):
21         """ Initializer """
22         # Call the parent class initializer
23         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Sprite Example")
24
25         # Sprite lists
26         self.player_list = None
27         self.wall_list = None
28
29         # Set up the player
30         self.player_sprite = None
31
32         # This variable holds our simple "physics engine"
33         self.physics_engine = None

```

(continues on next page)

(continued from previous page)

```

34
35     # Manage the view port
36     self.view_left = 0
37     self.view_bottom = 0
38
39 def setup(self):
40
41     # Set the background color
42     arcade.set_background_color(arcade.color.AMAZON)
43
44     # Reset the view port
45     self.view_left = 0
46     self.view_bottom = 0
47
48     # Sprite lists
49     self.player_list = arcade.SpriteList()
50     self.wall_list = arcade.SpriteList()
51
52     # Reset the score
53     self.score = 0
54
55     # Create the player
56     self.player_sprite = arcade.Sprite("images/character.png", SPRITE_SCALING_
57 →PLAYER)
57     self.player_sprite.center_x = 50
58     self.player_sprite.center_y = 64
59     self.player_list.append(self.player_sprite)
60
61     # --- Manually place walls
62
63     # Manually create and position a box at 300, 200
64     wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
65     wall.center_x = 300
66     wall.center_y = 200
67     self.wall_list.append(wall)
68
69     # Manually create and position a box at 364, 200
70     wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
71     wall.center_x = 364
72     wall.center_y = 200
73     self.wall_list.append(wall)
74
75     # --- Place boxes inside a loop
76     for x in range(173, 650, 64):
77         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
78         wall.center_x = x
79         wall.center_y = 350
80         self.wall_list.append(wall)
81
82     # --- Place walls with a list
83     coordinate_list = [[400, 500],
84                         [470, 500],
85                         [400, 570],
86                         [470, 570]]
87
88     # Loop through coordinates
89     for coordinate in coordinate_list:

```

(continues on next page)

(continued from previous page)

```

90         wall = arcade.Sprite("images/boxCrate_double.png", SPRITE_SCALING_BOX)
91         wall.center_x = coordinate[0]
92         wall.center_y = coordinate[1]
93         self.wall_list.append(wall)

94
95         # Create the physics engine. Give it a reference to the player, and
96         # the walls we can't run into.
97         self.physics_engine = arcade.PhysicsEngineSimple(self.player_sprite, self.
98             ←wall_list)

99     def on_draw(self):
100         arcade.start_render()
101         self.wall_list.draw()
102         self.player_list.draw()

103    def update(self, delta_time):
104        self.physics_engine.update()

105
106        # --- Manage Scrolling ---
107
108        # Track if we need to change the viewport
109
110        changed = False

111
112        # Scroll left
113        left_bndry = self.view_left + VIEWPORT_MARGIN
114        if self.player_sprite.left < left_bndry:
115            self.view_left -= left_bndry - self.player_sprite.left
116            changed = True

117
118        # Scroll right
119        right_bndry = self.view_left + SCREEN_WIDTH - VIEWPORT_MARGIN
120        if self.player_sprite.right > right_bndry:
121            self.view_left += self.player_sprite.right - right_bndry
122            changed = True

123
124        # Scroll up
125        top_bndry = self.view_bottom + SCREEN_HEIGHT - VIEWPORT_MARGIN
126        if self.player_sprite.top > top_bndry:
127            self.view_bottom += self.player_sprite.top - top_bndry
128            changed = True

129
130        # Scroll down
131        bottom_bndry = self.view_bottom + VIEWPORT_MARGIN
132        if self.player_sprite.bottom < bottom_bndry:
133            self.view_bottom -= bottom_bndry - self.player_sprite.bottom
134            changed = True

135
136        self.view_left = int(self.view_left)
137        self.view_bottom = int(self.view_bottom)

138
139        if changed:
140            arcade.set_viewport(self.view_left,
141                               SCREEN_WIDTH + self.view_left - 1,
142                               self.view_bottom,
143                               SCREEN_HEIGHT + self.view_bottom - 1)

```

(continues on next page)

(continued from previous page)

```

146 def on_key_press(self, key, modifiers):
147     """Called whenever a key is pressed. """
148
149     if key == arcade.key.UP:
150         self.player_sprite.change_y = MOVEMENT_SPEED
151     elif key == arcade.key.DOWN:
152         self.player_sprite.change_y = -MOVEMENT_SPEED
153     elif key == arcade.key.LEFT:
154         self.player_sprite.change_x = -MOVEMENT_SPEED
155     elif key == arcade.key.RIGHT:
156         self.player_sprite.change_x = MOVEMENT_SPEED
157
158 def on_key_release(self, key, modifiers):
159     """Called when the user releases a key. """
160
161     if key == arcade.key.UP or key == arcade.key.DOWN:
162         self.player_sprite.change_y = 0
163     elif key == arcade.key.LEFT or key == arcade.key.RIGHT:
164         self.player_sprite.change_x = 0
165
166
167 def main():
168     """ Main method """
169     window = MyGame()
170     window.setup()
171     arcade.run()
172
173
174 if __name__ == "__main__":
175     main()

```

1.23 Libraries and Modules



A *library* is a collection of code for functions and classes. Often, these libraries are written by someone else and

brought into the project so that the programmer does not have to “reinvent the wheel.” In Python the term used to describe a library of code is module.

By using `import arcade` and `import random`, the programs created so far have already used modules. A library can be made up of multiple modules that can be imported. Often a library only has one module, so these words can sometimes be used interchangeably.

Modules are often organized into groups of similar functionality. In this class programs have already used functions from the `math` module, the `random` module, and the `arcade` library. Modules can be organized so that individual modules contain other modules. For example, the `arcade` module contains submodules for `arcade.key`, and `arcade.color`.

Modules are not loaded unless the program asks them to. This saves time and computer memory. This chapter shows how to create a module, and how to import and use that module.

1.23.1 Why Create a Library?

There are three major reasons for a programmer to create his or her own libraries:

1. It breaks the code into smaller, easier to use parts.
2. It allows multiple people to work on a program at the same time.
3. The code written can be easily shared with other programmers.

Some of the programs already created in this book have started to get rather long. By separating a large program into several smaller programs, it is easier to manage the code. For example, in the prior chapter’s sprite example, a programmer could move the sprite class into a separate file. In a complex program, each sprite might be contained in its own file.

If multiple programmers work on the same project, it is nearly impossible to do so if all the code is in one file. However, by breaking the program into multiple pieces, it becomes easier. One programmer could work on developing an “Orc” sprite class. Another programmer could work on the “Goblin” sprite class. Since the sprites are in separate files, the programmers do not run into conflict.

Modern programmers rarely build programs from scratch. Often programs are built from parts of other programs that share the same functionality. If one programmer creates code that can handle a mortgage application form, that code will ideally go into a library. Then any other program that needs to manage a mortgage application form at that bank can call on that library.

1.23.2 Creating Your Own Module/Library File

Video: Libraries

In this example we will break apart a short program into multiple files. Here we have a function in a file named `test.py`, and a call to that function:

Listing 89: `test.py` with everything in it

```
1 # Foo function
2 def foo():
3     print("foo!")
4
5 def main():
6     # Foo call
7     foo()
8
9 main()
```

Yes, this program is not too long to be in one file. But if both the function and the main program code were long, it would be different. If we had several functions, each 100 lines long, it would be time consuming to manage that large of a file. But for this example we will keep the code short for clarity.

We can move the `foo` function out of this file. Then this file would be left with only the main program code. (In this example there is no reason to separate them, aside from learning how to do so.)

To do this, create a new file and copy the `foo` function into it. Save the new file with the name `my_functions.py`. The file must be saved to the same directory as `test.py`.

Listing 90: `my_functions.py`

```
1 # Foo function
2 def foo():
3     print("foo!")
```

Listing 91: `test.py` that doesn't work

```
1 def main():
2     # Foo call that doesn't work
3     foo()
4
5 main()
```

Unfortunately it isn't as simple as this. The file `test.py` does not know to go and look at the `my_functions.py` file and import it. We have to add the command to import it:

Listing 92: `test.py` that imports but still doesn't work

```
1 # Import the my_functions.py file
2 import my_functions
3
4 def main():
5     # Foo call that still doesn't work
6     foo()
7
8 main()
```

That still doesn't work. What are we missing? Just like when we import `arcade`, we have to put the package name in front of the function. Like this:

Listing 93: `test.py` that finally works

```
1 # Import the my_functions.py file
2 import my_functions
3
4 def main():
5     # Foo call that does work
6     my_functions.foo()
7
8 main()
```

This works because `my_functions.` is prepended to the function call.

1.23.3 Namespace

Video: Namespace

A program might have two library files that need to be used. What if the libraries had functions that were named the same? What if there were two functions named `print_report`, one that printed grades, and one that printed an account statement? For instance:

Listing 94: `student_functions.py`

```
1 def print_report():
2     print("Student Grade Report: ")
```

Listing 95: `financial_functions.py`

```
1 def print_report():
2     print("Financial Report: ")
```

How do you get a program to specify which function to call? Well, that is pretty easy. You specify the *namespace*. The namespace is the work that appears before the function name in the code below:

Listing 96: `test.py` that calls different `print_report` functions

```
1 import student_functions
2 import financial_functions
3
4 def main():
5     student_functions.print_report()
6     financial_functions.print_report()
7
8 main()
```

So now we can see why this might be needed. But what if you don't have name collisions? Typing in a namespace each and every time can be tiresome. You can get around this by importing the library into the *local namespace*. The local namespace is a list of functions, variables, and classes that you don't have to prepend with a namespace. Going back to the `foo` example, let's remove the original import and replace it with a new type of import:

Listing 97: `test.py`

```
1 # import foo
2 from my_functions import *
3
4 def main():
5     foo()
6
7 main()
```

This works even without `my_functions.` prepended to the function call. The asterisk is a wildcard that will import all functions from `my_functions`. A programmer could import individual ones if desired by specifying the function name.

1.23.4 Third Party Libraries

When working with Python, it is possible to use many libraries that are built into Python. Take a look at all the libraries that are available here:

<http://docs.python.org/3/py-modindex.html>

It is possible to download and install other libraries. There are libraries that work with the web, complex numbers, databases, and more.

- Arcade: The library that this book uses to create games. <http://arcade.academy>
- Pygame: Another library used to create games, and the inspiration behind the creation of the Arcade library. <http://www.pygame.org/docs/>
- Pymunk: A great library for running 2D physics. Also works with Arcade, see these examples <http://www.pymunk.org/>
- wxPython: Create GUI programs, with windows, menus, and more. <http://www.wxpython.org/>
- pydot: Generate complex directed and non-directed graphs <http://code.google.com/p/pydot/>
- NumPy: Sophisticated library for working with matrices. <http://numpy.scipy.org/>
- Pandas: A library for data analysis. <https://pandas.pydata.org/>
- Pillow: Work with images. <https://pillow.readthedocs.io/en/latest/>
- Pyglet: Another graphics library. Arcade uses this library. <https://bitbucket.org/pyglet/pyglet/wiki/Home>

You can do analysis and create your own interactive notebook using Jupyter:

- Jupyter: <http://jupyter.org/>

Some libraries we give examples of in this chapter:

- OpenPyXL: A library for reading and writing Excel files. <https://openpyxl.readthedocs.io/en/stable/>
- Beautiful Soup: Grab data off websites, and create your own web bots. <https://www.crummy.com/software/BeautifulSoup/>
- Matplotlib: Plot data automatically: <https://matplotlib.org/>

A wonderful list of Python libraries and links to installers for them is available here:

You can search up some top packages/libraries and stand alone projects to get an idea of what you can do. There are many articles like [Top 15 Python Libraries for Data Science in 2017](#).

Examples: OpenPyXL Library

This example uses a library called OpenPyXL to write an Excel file. It is also easy to read from an Excel file. You can install OpenPyXL from the Windows command prompt by typing `pip install openpyxl`. If you are on the Mac or a Linux machine, you can type `sudo pip3 install openpyxl`.

Note: When starting the command prompt, you might need to right-click on it and select “Run as administrator” if you get permission errors when installing. And if you are working on a lab computer, you might not have permission to install libraries.

Listing 98: openpyxl_example.py

```

1  """
2  Example using OpenPyXL to create an Excel worksheet
3  """
4
5  from openpyxl import Workbook
6  import random
7
8  # Create an Excel workbook
9  work_book = Workbook()
10

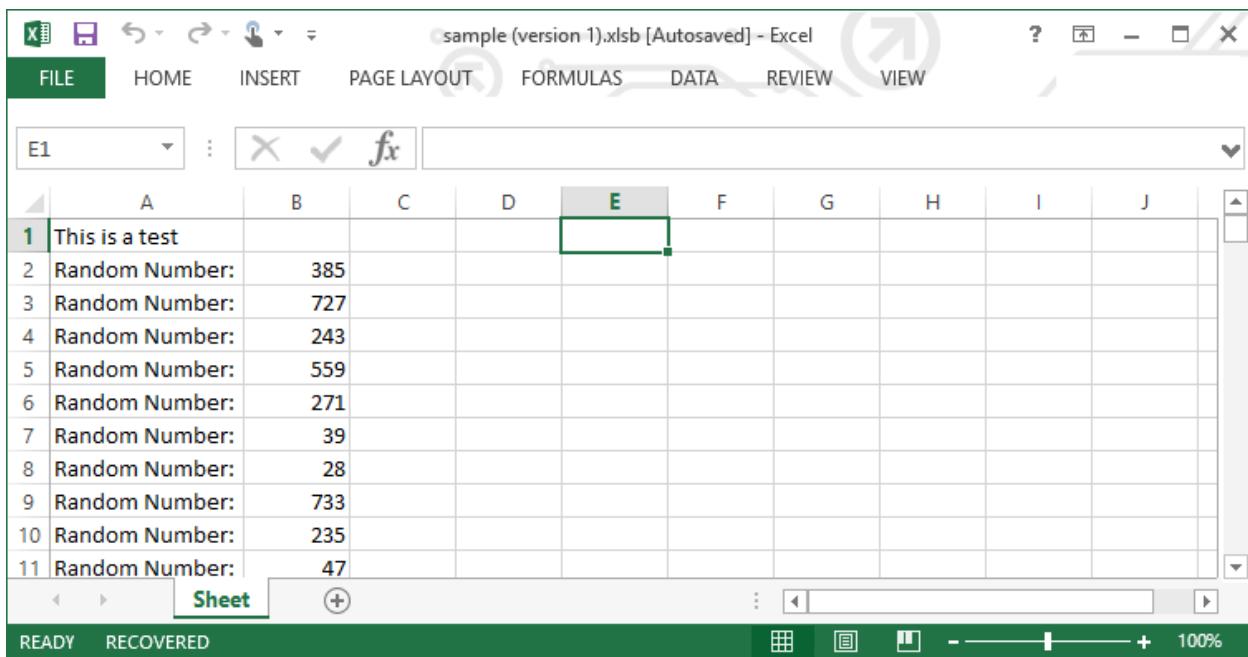
```

(continues on next page)

(continued from previous page)

```
11 # Grab the active worksheet
12 work_sheet = work_book.active
13
14 # Data can be assigned directly to cells
15 work_sheet['A1'] = "This is a test"
16
17 # Rows can also be appended
18 for i in range(200):
19     work_sheet.append(["Random Number:", random.randrange(1000)])
20
21 # Save the file
22 work_book.save("sample.xlsx")
```

The output of this program is an Excel file:



Examples: Beautiful Soup Library

This example grabs information off a web page. You can install Beautiful Soup from the Windows command prompt by typing `pip install bs4`. If you are on the Mac or a Linux machine, you can type `sudo pip3 install bs4`.

Listing 99: bs4_example.py

```
1 """
2 Example showing how to read in from a web page
3 """
4
5 from bs4 import BeautifulSoup
6 import urllib.request
7
8 # Read in the web page
9 url_address = "http://simpson.edu"
```

(continues on next page)

(continued from previous page)

```

10 page = urllib.request.urlopen(url_address)
11
12 # Parse the web page
13 soup = BeautifulSoup(page.read(), "html.parser")
14
15 # Get a list of level 1 headings in the page
16 headings = soup.findAll("h1")
17
18 # Loop through each row
19 for heading in headings:
20     print(heading.text)

```

Examples: Matplotlib Library

Here is an example of what you can do with the third party library “Matplotlib.” You can install Matplotlib from the Windows command prompt by typing pip install matplotlib. If you are on the Mac or a Linux machine, you can type pip3 install matplotlib.

Example 1: Simple Plot

To start with, here is the code to create a simple line chart with four values:

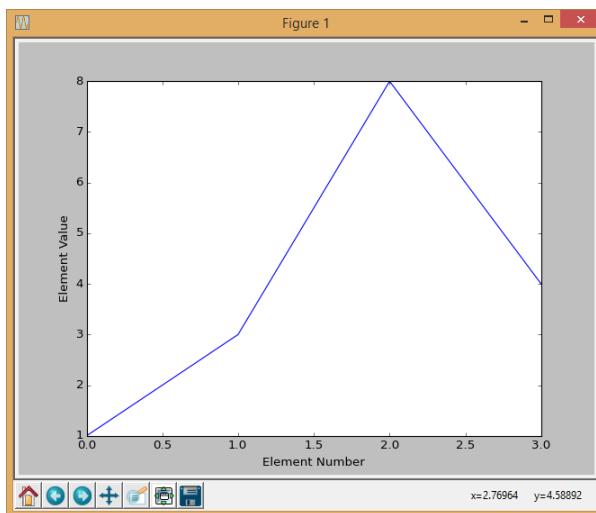


Fig. 42: Simple Line Graph

Listing 100: Example 1

```

1 """
2 Line chart with four values.
3 The x-axis defaults to start at zero.
4 """
5 import matplotlib.pyplot as plt
6
7 y = [1, 3, 8, 4]
8

```

(continues on next page)

(continued from previous page)

```

9 plt.plot(y)
10 plt.ylabel('Element Value')
11 plt.xlabel('Element Number')
12
13 plt.show()

```

Note that you can zoom in, pan, and save the graph. You can even save the graph in vector formats like ps and svg that import into documents without loss of quality like raster graphics would have.

Example 2: Specify x Values

The x value for Example 1, defaults to start at zero. You can change this default and specify your own x values to go with the y values. See Example 2 below.

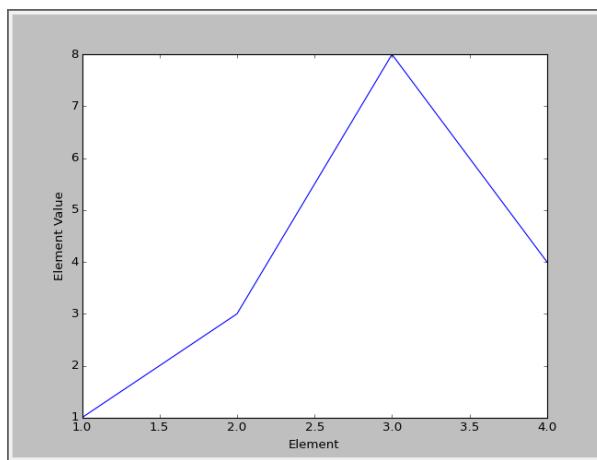


Fig. 43: Specifying the x values

Listing 101: Example 2

```

1 """
2 Line chart with four values.
3 The x-axis values are specified as well.
4 """
5 import matplotlib.pyplot as plt
6
7 x = [1, 2, 3, 4]
8 y = [1, 3, 8, 4]
9
10 plt.plot(x, y)
11
12 plt.ylabel('Element Value')
13 plt.xlabel('Element')
14
15 plt.show()

```

Example 3: Add A Second Data Series

It is trivial to add another data series to the graph.

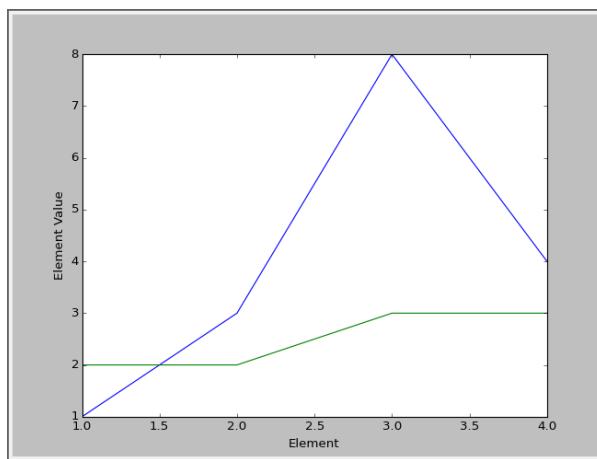


Fig. 44: Graphing two data series

Listing 102: Example 3

```

1 """
2 This example shows graphing two different series
3 on the same graph.
4 """
5 import matplotlib.pyplot as plt
6
7 x = [1, 2, 3, 4]
8 y1 = [1, 3, 8, 4]
9 y2 = [2, 2, 3, 3]
10
11 plt.plot(x, y1)
12 plt.plot(x, y2)
13
14 plt.ylabel('Element Value')
15 plt.xlabel('Element')
16
17 plt.show()
18

```

Example 4: Add A Legend

You can add a legend to the graph:

Listing 103: Example 4

```

1
2 import matplotlib.pyplot as plt
3
4 x = [1, 2, 3, 4]
5 y1 = [1, 3, 8, 4]
6 y2 = [2, 2, 3, 3]
7
8 plt.plot(x, y1, label = "Series 1")
9 plt.plot(x, y2, label = "Series 2")

```

(continues on next page)

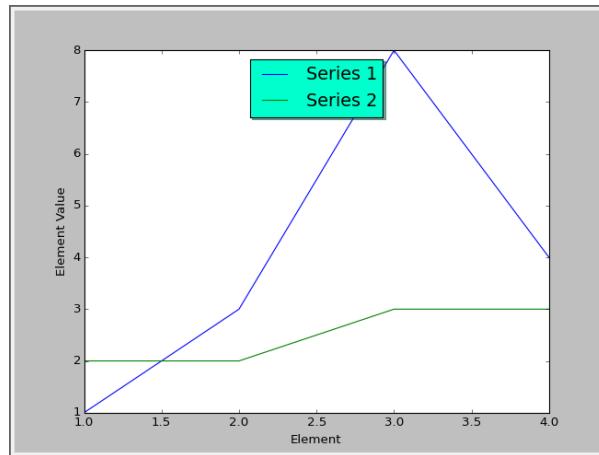


Fig. 45: Adding a legend

(continued from previous page)

```

10
11 legend = plt.legend(loc='upper center', shadow=True, fontsize='x-large')
12 legend.get_frame().set_facecolor('#00FFCC')
13
14 plt.ylabel('Element Value')
15 plt.xlabel('Element')
16
17 plt.show()

```

Example 5: Add Annotations

You can add annotations to a graph:

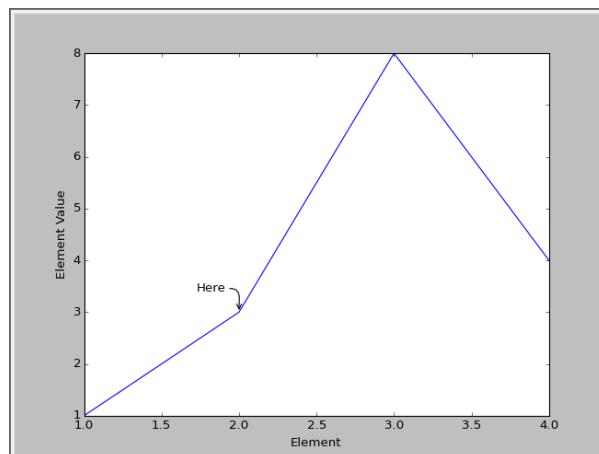


Fig. 46: Adding annotations

Listing 104: Example 5

```

1 """
2 Annotating a graph
3 """
4 import matplotlib.pyplot as plt
5
6 x = [1, 2, 3, 4]
7 y = [1, 3, 8, 4]
8
9 plt.annotate('Here',
10             xy=(2, 3),
11             xycoords='data',
12             xytext=(-40, 20),
13             textcoords='offset points',
14             arrowprops=dict(arrowstyle="->",
15                             connectionstyle="arc,angleA=0,armA=30,rad=10"),
16             )
17
18 plt.plot(x, y)
19
20 plt.ylabel('Element Value')
21 plt.xlabel('Element')
22
23 plt.show()

```

Example 6: Change Line Styles

Don't like the default line styles for the graph? That can be changed by adding a third parameter to the plot command.

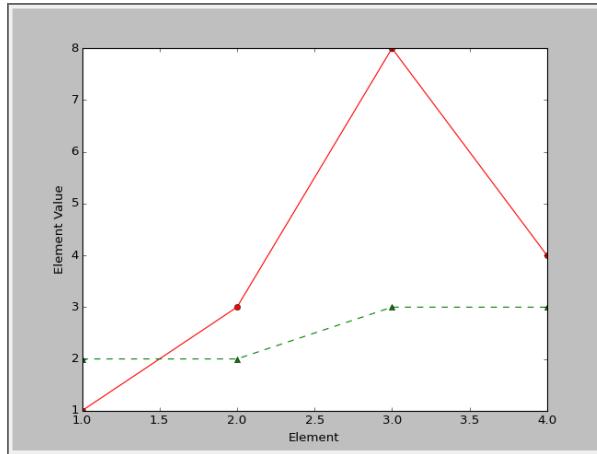


Fig. 47: Specifying the line style

Listing 105: Example 6

```

1 """
2 This shows how to set line style and markers.
3 """
4 import matplotlib.pyplot as plt

```

(continues on next page)

(continued from previous page)

```

5
6 x = [1, 2, 3, 4]
7 y1 = [1, 3, 8, 4]
8 y2 = [2, 2, 3, 3]
9
10 # First character: Line style
11 # One of '-', '--', '-.', ':', 'None', ' ', "
12
13 # Second character: color
14 # http://matplotlib.org/1.4.2/api/colors_api.html
15
16 # Third character: marker shape
17 # http://matplotlib.org/1.4.2/api/markers_api.html
18
19 plt.plot(x, y1, '-ro')
20 plt.plot(x, y2, '--g^')
21
22 plt.ylabel('Element Value')
23 plt.xlabel('Element')
24
25 plt.show()

```

Example 7: Bar Chart

A bar chart is as easy as changing plot to bar.

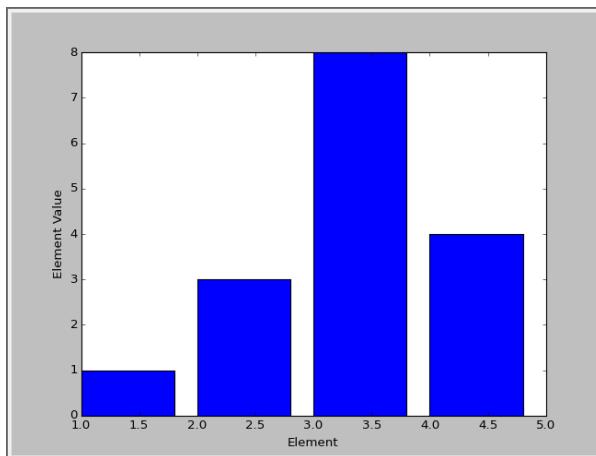


Fig. 48: Bar chart

Listing 106: Example 7

```

1 """
2 How to do a bar chart.
3 """
4 import matplotlib.pyplot as plt
5
6 x = [1, 2, 3, 4]
7 y = [1, 3, 8, 4]
8

```

(continues on next page)

(continued from previous page)

```

9 plt.bar(x, y)
10
11 plt.ylabel('Element Value')
12 plt.xlabel('Element')
13
14 plt.show()

```

Example 8: Axis Labels

You can add labels to axis values.

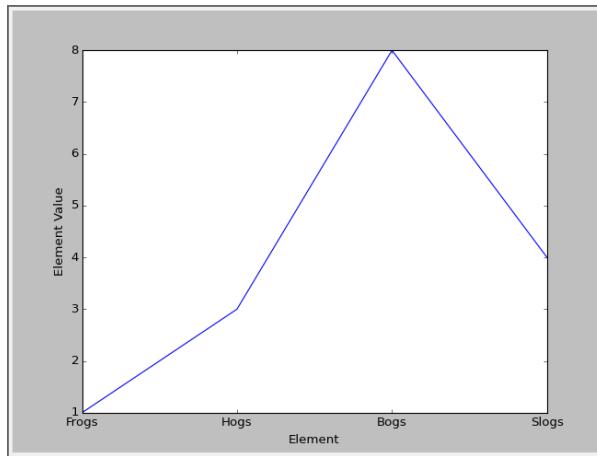


Fig. 49: X-axis labels

Listing 107: Example 8

```

1 """
2 How to add x axis value labels.
3 """
4 import matplotlib.pyplot as plt
5
6 x = [1, 2, 3, 4]
7 y = [1, 3, 8, 4]
8
9 plt.plot(x, y)
10
11 labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']
12 plt.xticks(x, labels)
13
14 plt.ylabel('Element Value')
15 plt.xlabel('Element')
16
17 plt.show()

```

Example 9: Graph Functions

You can graph functions as well. This uses a different package called numpy to graph a sine function.

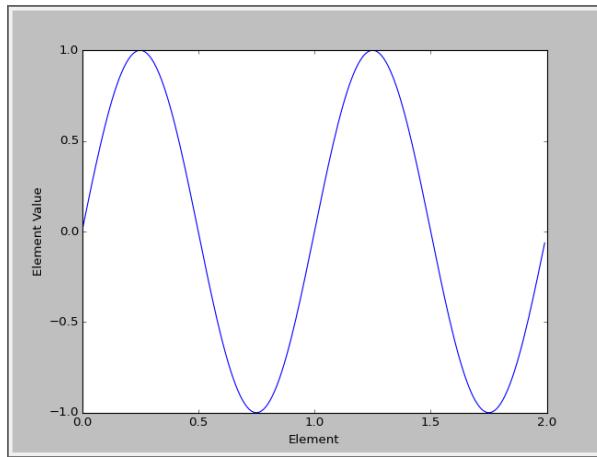


Fig. 50: Graphing a sine function

Listing 108: Example 9

```

1 """
2 Using the numpy package to graph a function over
3 a range of values.
4 """
5 import numpy
6 import matplotlib.pyplot as plt
7
8 x = numpy.arange(0.0, 2.0, 0.001)
9 y = numpy.sin(2 * numpy.pi * x)
10
11 plt.plot(x, y)
12
13 plt.ylabel('Element Value')
14 plt.xlabel('Element')
15
16 plt.show()

```

Example 10: Graph Functions With Fill

You can fill in a graph if you like.

Listing 109: Example 10

```

1 """
2 Using 'fill' to fill in a graph
3 """
4 import numpy
5 import matplotlib.pyplot as plt
6
7 x = numpy.arange(0.0, 2.0, 0.001)
8 y = numpy.sin(2 * numpy.pi * x)
9
10 plt.plot(x, y)
11

```

(continues on next page)

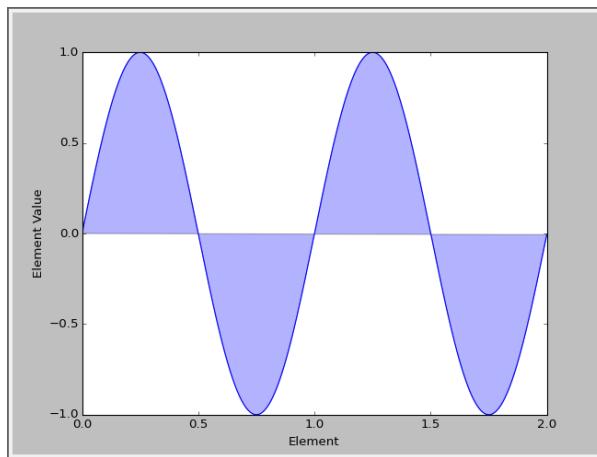


Fig. 51: Filling in a graph

(continued from previous page)

```

12 # 'b' means blue. 'alpha' is the transparency.
13 plt.fill(x, y, 'b', alpha=0.3)
14
15 plt.ylabel('Element Value')
16 plt.xlabel('Element')
17
18 plt.show()

```

Example 11: Pie Chart

Create a pie chart.

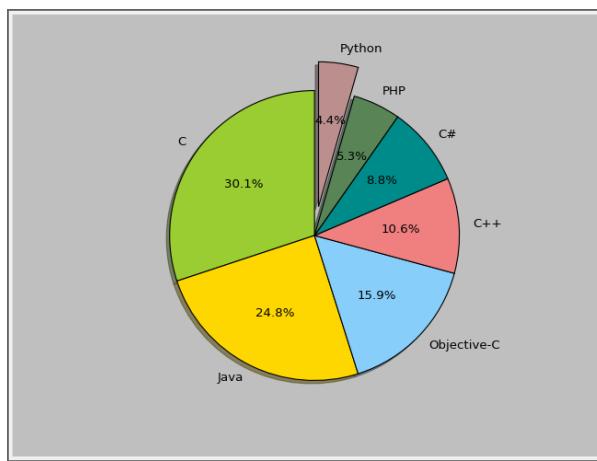


Fig. 52: Pie chart

Listing 110: Example 11

```

1 """
2 Create a pie chart

```

(continues on next page)

(continued from previous page)

```

3 """
4 import matplotlib.pyplot as plt
5
6 # Labels for the pie chart
7 labels = ['C', 'Java', 'Objective-C', 'C++', 'C#', 'PHP', 'Python']
8
9 # Sizes for each label. We use this to make a percent
10 sizes = [17, 14, 9, 6, 5, 3, 2.5]
11
12 # For list of colors, see:
13 # https://matplotlib.org/examples/color/named_colors.html
14 colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral', 'darkcyan', 'aquamarine',
15           'rosybrown']
16
17 # How far out to pull a slice. Normally zero.
18 explode = (0, 0.0, 0, 0, 0, 0, 0.2)
19
20 # Set aspect ratio to be equal so that pie is drawn as a circle.
21 plt.axis('equal')
22
23 # Finally, plot the chart
24 plt.pie(sizes, explode=explode, labels=labels, colors=colors,
25          autopct='%.1f%%', shadow=True, startangle=90)
26 plt.show()

```

Example 12: Candlestick Chart

You can do really fancy things, like pull stock data from the web and create a candlestick graph for Apple Computer:

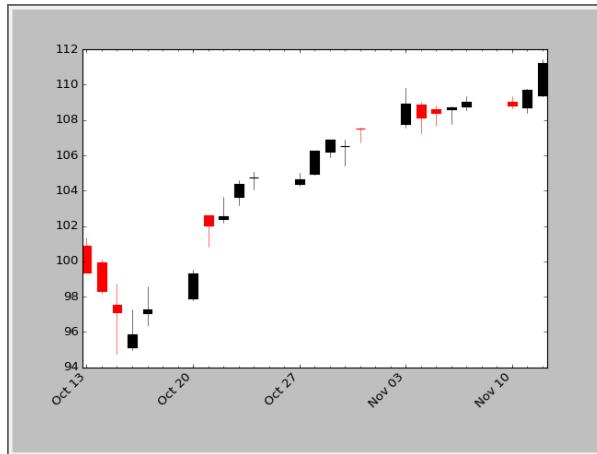


Fig. 53: Candlestick chart

Listing 111: Example 12

```

1 """
2 Create a candlestick chart for a stock
3 """
4 import matplotlib.pyplot as plt

```

(continues on next page)

(continued from previous page)

```

5   from matplotlib.dates import DateFormatter, WeekdayLocator, \
6       DayLocator, MONDAY
7   from matplotlib.finance import quotes_historical_yahoo_ohlc, candlestick_ohlc
8
9   # Grab the stock data between these dates
10  date1 = (2014, 10, 13)
11  date2 = (2014, 11, 13)
12
13  # Go to the web and pull the stock info
14  quotes = quotes_historical_yahoo_ohlc('AAPL', date1, date2)
15  if len(quotes) == 0:
16      raise SystemExit
17
18  # Set up the graph
19  fig, ax = plt.subplots()
20  fig.subplots_adjust(bottom=0.2)
21
22  # Major ticks on Mondays
23  mondays = WeekdayLocator(MONDAY)
24  ax.xaxis.set_major_locator(mondays)
25
26  # Minor ticks on all days
27  alldays = DayLocator()
28  ax.xaxis.set_minor_locator(alldays)
29
30  # Format the days
31  weekFormatter = DateFormatter('%b %d') # e.g., Jan 12
32  ax.xaxis.set_major_formatter(weekFormatter)
33  ax.xaxis_date()
34
35  candlestick_ohlc(ax, quotes, width=0.6)
36
37  ax.autoscale_view()
38  plt.setp(plt.gca().get_xticklabels(), rotation=45, horizontalalignment='right')
39
40  plt.show()

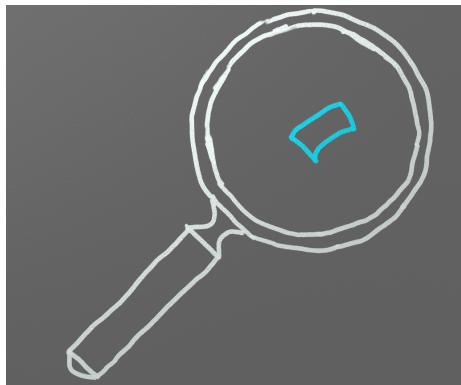
```

There are many more things that can be done with matplotlib. Take a look at the thumbnail gallery:

<http://matplotlib.org/gallery.html>

1.24 Searching

Searching is an important and very common operation that computers do all the time. Searches are used every time someone does a ctrl-f for “find”, when a user uses “type-to” to quickly select an item, or when a web server pulls information about a customer to present a customized web page with the customer’s order.



There are a lot of ways to search for data. Google has based an entire multi-billion dollar company on this fact. This chapter introduces the two simplest methods for searching, the linear search and the binary search.

1.24.1 Reading From a File

Before discussing how to search we need to learn how to read data from a file. Reading in a data set from a file is way more fun than typing it in by hand each time.

Let's say we need to create a program that will allow us to quickly find the name of a super-villain. To start with, our program needs a database of super-villains. To download this data set, download and save this file:

`super_villains.txt`

These are random names generated by the nine.frenchboys.net website, although last I checked they no longer have a super-villain generator. They have other cool random name generators though.

Save this file and remember which directory you saved it to.

In the same directory as `super_villains.txt`, create, save, and run the following Python program:

Listing 112: Read in a file

```
1 def main():
2     """ Read in lines from a file """
3
4     # Open the file for reading, and store a pointer to it in the new
5     # variable "file"
6     my_file = open("super_villains.txt")
7
8     # Loop through each line in the file like a list
9     for line in my_file:
10         print(line)
11
12
13 main()
```

There is only one new command in this code: `open`. Because it is a built-in function like `print`, there is no need for an `import`. Full details on this function can be found in the [Python documentation](#) but at this point the documentation for that command is so technical it might not even be worth looking at.

The above program has two problems with it, but it provides a simple example of reading in a file. Line 6 opens a file and gets it ready to be read. The name of the file is in between the quotes. The new variable `my_file` is an object that represents the file being read. Line 9 shows how a normal `for` loop may be used to read through a file line by line. Think of the file as a list of lines, and the new variable `line` will be set to each of those lines as the program runs through the loop.

Try running the program. One of the problems with the it is that the text is printed double-spaced. The reason for this is that each line pulled out of the file and stored in the variable line includes the carriage return as part of the string. Remember the carriage return and line feed introduced back in Chapter 1? The `print` statement adds yet another carriage return and the result is double-spaced output.

The second problem is that the file is opened, but not closed. This problem isn't as obvious as the double-spacing issue, but it is important. The Windows operating system can only open so many files at once. A file can normally only be opened by one program at a time. Leaving a file open will limit what other programs can do with the file and take up system resources. It is necessary to close the file to let Windows know the program is no longer working with that file. In this case it is not too important because once any program is done running, the Windows will automatically close any files left open. But since it is a bad habit to program like that, let's update the code:

Listing 113: Read in a file

```

1 def main():
2     """ Read in lines from a file """
3
4     # Open the file for reading, and store a pointer to it in the new
5     # variable "file"
6     my_file = open("super_villains.txt")
7
8     # Loop through each line in the file like a list
9     for line in my_file:
10         # Remove any line feed, carriage returns or spaces at the end of the line
11         line = line.strip()
12         print(line)
13
14     my_file.close()
15
16
17 main()

```

The listing above works better. It has two new additions. On line 4 is a call to the `strip` method built into every `String` class. This function returns a new string without the trailing spaces and carriage returns of the original string. The method does not alter the original string but instead creates a new one. This line of code would not work:

```
line.strip()
```

If the programmer wants the original variable to reference the new string, she must assign it to the new returned string as shown on line 4.

The second addition is on line 7. This closes the file so that the operating system doesn't have to go around later and clean up open files after the program ends.

1.24.2 Reading Into an Array

It is useful to read in the contents of a file to an array so that the program can do processing on it later. This can easily be done in python with the following code:

Listing 114: Read in a file from disk and put it in an array

```

1 def main():
2     """ Read in lines from a file """
3
4     # Open the file for reading, and store a pointer to it in the new
5     # variable "file"

```

(continues on next page)

(continued from previous page)

```

6 my_file = open("super_villains.txt")
7
8 # Create an empty list to store our names
9 name_list = []
10
11 # Loop through each line in the file like a list
12 for line in my_file:
13     # Remove any line feed, carriage returns or spaces at the end of the line
14     line = line.strip()
15
16     # Add the name to the list
17     name_list.append(line)
18
19 my_file.close()
20
21 print("There were", len(name_list), "names in the file.")
22
23
24 main()

```

This combines the new pattern of how to read a file, along with the previously learned pattern of how to create an empty array and append to it as new data comes in, which was shown back in [Adding to a List](#). To verify the file was read into the array correctly a programmer could print the length of the array:

```
print("There were", len(name_list), "names in the file.")
```

Or the programmer could print the entire contents of the array:

```
for name in name_list:
    print(name)
```

Go ahead and make sure you can read in the file before continuing on to the different searches.

1.24.3 Linear Search

If a program has a set of data in an array, how can it go about finding where a specific element is? This can be done one of two ways. The first method is to use a *linear search*. This starts at the first element, and keeps comparing elements until it finds the desired element (or runs out of elements.)

1.24.4 Linear Search Algorithm

Listing 115: Linear search

```

1 # --- Linear search
2 key = "Morgiana the Shrew"
3
4 # Start at the beginning of the list
5 current_list_position = 0
6
7 # Loop until you reach the end of the list, or the value at the
8 # current position is equal to the key
9 while current_list_position < len(name_list) and name_list[current_list_position] != key:

```

(continues on next page)

(continued from previous page)

```

10     # Advance to the next item in the list
11     current_list_position += 1
12
13
14 if current_list_position < len(name_list):
15     print("The name is at position", current_list_position)
16 else:
17     print("The name was not in the list.")

```

The linear search is rather simple. Line 5 sets up an increment variable that will keep track of exactly where in the list the program needs to check next. The first element that needs to be checked is zero, so `current_list_position` is set to zero.

The next line is a bit more complex. The computer needs to keep looping until one of two things happens. It finds the element, or it runs out of elements. The first comparison sees if the current element we are checking is less than the length of the list. If so, we can keep looping. The second comparison sees if the current element in the name list is equal to the name we are searching for.

This check to see if the program has run out of elements *must occur first*. Otherwise the program will check against a non-existent element which will cause an error.

Line 12 simply moves to the next element if the conditions to keep searching are met in line 9.

At the end of the loop, the program checks to see if the end of the list was reached on line 14. Remember, a list of n elements is numbered 0 to n-1. Therefore if i is equal to the length of the list, the end has been reached. If it is less, we found the element.

The full example with both the reading in the file and the search is below:

Listing 116: Linear search

```

1 def main():
2     """ Read in lines from a file """
3
4     # Open the file for reading, and store a pointer to it in the new
5     # variable "file"
6     my_file = open("super_villains.txt")
7
8     # Create an empty list to store our names
9     name_list = []
10
11    # Loop through each line in the file like a list
12    for line in my_file:
13        # Remove any line feed, carriage returns or spaces at the end of the line
14        line = line.strip()
15
16        # Add the name to the list
17        name_list.append(line)
18
19    my_file.close()
20
21    print("There were", len(name_list), "names in the file.")
22
23    # --- Linear search
24    key = "Morgiana the Shrew"
25
26    # Start at the beginning of the list
27    current_list_position = 0

```

(continues on next page)

(continued from previous page)

```

28
29     # Loop until you reach the end of the list, or the value at the
30     # current position is equal to the key
31     while current_list_position < len(name_list) and name_list[current_list_position] ↴
32         != key:
33         # Advance to the next item in the list
34         current_list_position += 1
35
36     if current_list_position < len(name_list):
37         print("The name is at position", current_list_position)
38     else:
39         print("The name was not in the list.")
40
41 main()

```

We can improve on this example by moving both the reading of the file and the search into their own functions:

Listing 117: Linear search

```

1 def read_in_file(file_name):
2     """ Read in lines from a file """
3
4     # Open the file for reading, and store a pointer to it in the new
5     # variable "file"
6     my_file = open(file_name)
7
8     # Create an empty list to store our names
9     name_list = []
10
11    # Loop through each line in the file like a list
12    for line in my_file:
13        # Remove any line feed, carriage returns or spaces at the end of the line
14        line = line.strip()
15
16        # Add the name to the list
17        name_list.append(line)
18
19    my_file.close()
20
21    return name_list
22
23
24 def linear_search(key, name_list):
25     """ Linear search """
26
27     # Start at the beginning of the list
28     current_list_position = 0
29
30     # Loop until you reach the end of the list, or the value at the
31     # current position is equal to the key
32     while current_list_position < len(name_list) and name_list[current_list_position] ↴
33         != key:
34
35         # Advance to the next item in the list
36         current_list_position += 1

```

(continues on next page)

(continued from previous page)

```

37     return current_list_position
38
39
40 def main():
41
42     key = "Morgiana the Shrew"
43     name_list = read_in_file("super_villains.txt")
44     list_position = linear_search(key, name_list)
45
46     if list_position < len(name_list):
47         print("The name", key, "is at position", list_position)
48     else:
49         print("The name", key, "was not in the list.")
50
51
52 main()

```

1.24.5 Variations On The Linear Search

Variations on the linear search can be used to create several common algorithms. Specifically, you can use it to see if *any* items in a list match a property, or if *all* items match a property. You can also use it to pull all matching items out of a list.

Does At Least One Item Have a Property?

For example, here is a function that uses the linear search to see if there is an item in `my_list` that matches the value in `key`.

Listing 118: `check_if_one_item_has_property_v1`

```

1 def check_if_one_item_has_property_v1(my_list, key):
2     """
3         Return true if at least one item has a
4         property.
5     """
6     list_position = 0
7     while list_position < len(my_list) and my_list[list_position] != key:
8         list_position += 1
9
10    if list_position < len(my_list):
11        # Found an item with the property
12        return True
13    else:
14        # There is no item with the property
15        return False

```

Using the `break` statement, which exits a loop early, we can simplify the code:

Listing 119: `check_if_one_item_has_property_v2`

```

1 def check_if_one_item_has_property_v2(my_list, key):
2     """
3         Return true if at least one item has a
4         property.

```

(continues on next page)

(continued from previous page)

```

5     """
6     for item in my_list:
7         if item == key:
8             # Found an item that matched. Return True
9             return True
10
11    # Went through the whole list. Return False.
12    return False

```

Do All Items Have a Property?

How would you test to see if *all* items in a list match a property? We just need to reverse a few things in the algorithm above.

Listing 120: check_if_all_items_have_property

```

1 def check_if_all_items_have_property(my_list, key):
2     """
3     Return true if at ALL items have a property.
4     """
5     for item in my_list:
6         if item != key:
7             # Found an item that didn't match. Return False.
8             return False
9
10    # Got through the entire list. There were no mis-matches.
11    return True

```

Create a List With All Items Matching a Property

Another common operation is to grab all the items out of a list that match:

Listing 121: get_matching_items

```

1 def get_matching_items(my_list, key):
2     """
3     Build a brand new list that holds all the items
4     that match our property.
5     """
6     matching_list = []
7     for item in my_list:
8         if item == key:
9             matching_list.append(item)
10    return matching_list

```

Full Example

For a full example, see below:

Listing 122: linear_search_variations_2.py

```

1 import random
2
3
4 def create_list(list_size):
5     """ Create a list of random numbers """
6     my_list = []
7
8     for i in range(list_size):
9         my_list.append(random.randrange(100))
10
11    return my_list
12
13
14 def check_if_one_item_has_property_v1(my_list, key):
15     """
16     Return true if at least one item has a
17     property.
18     """
19     list_position = 0
20     while list_position < len(my_list) and my_list[list_position] != key:
21         list_position += 1
22
23     if list_position < len(my_list):
24         # Found an item with the property
25         return True
26     else:
27         # There is no item with the property
28         return False
29
30
31 def check_if_one_item_has_property_v2(my_list, key):
32     """
33     Return true if at least one item has a
34     property.
35     """
36     for item in my_list:
37         if item == key:
38             # Found an item that matched. Return True
39             return True
40
41     # Went through the whole list. Return False.
42     return False
43
44
45 def check_if_all_items_have_property(my_list, key):
46     """
47     Return true if at ALL items have a property.
48     """
49     for item in my_list:
50         if item != key:
51             # Found an item that didn't match. Return False.
52             return False
53
54     # Got through the entire list. There were no mis-matches.
55     return True

```

(continues on next page)

(continued from previous page)

```

56
57
58 def get_matching_items(my_list, key):
59     """
60         Build a brand new list that holds all the items
61         that match our property.
62     """
63     matching_list = []
64     for item in my_list:
65         if item == key:
66             matching_list.append(item)
67     return matching_list
68
69
70 def main():
71
72     # Create a list of 50 numbers
73     my_list = create_list(50)
74     print(my_list)
75
76     # Is at least one item zero?
77     key = 0
78     result = check_if_one_item_has_property_v1(my_list, 0)
79     if result:
80         print("At least one item in the list is", key)
81     else:
82         print("No item in the list is", key)
83
84     # Get items that match the key
85     matching_list = get_matching_items(my_list, key)
86     print("Matching items:", matching_list)
87
88     # Are all items matching?
89     result = check_if_all_items_have_property(my_list, key)
90     print("All items in random list matching?", result)
91
92     other_list = [0, 0, 0, 0, 0]
93     result = check_if_all_items_have_property(other_list, key)
94     print("All items in other list matching?", result)
95
96
97 main()

```

1.24.6 Variations On The Linear Search With Objects

For example, say we had a list of objects for our text adventure. We might want to check that list and see if any of the items are in the same room as our player. Or if all the items are. Or we might want to build a list of items that the user is carrying if they are all in a “special” room that represents the player’s inventory.

To begin with, we’d need to define our adventure object:

Listing 123: Adventure Object class

```

1 class AdventureObject:
2     """
3         Class that defines an object in a text adventure game """

```

(continues on next page)

(continued from previous page)

```

3     def __init__(self, description, room):
4         """ Constructor """
5
6
7         # Description of the object
8         self.description = description
9
10        # The number of the room that the object is in
11        self.room = room

```

Does At Least One Item Have a Property?

Is at least one object in the specified room? We can check.

Listing 124: Check if list has an item that has a property - while loop

```

1 def check_if_one_item_is_in_room_v1(my_list, room):
2     """
3         Return true if at least one item has a
4         property.
5     """
6     i = 0
7     while i < len(my_list) and my_list[i].room != room:
8         i += 1
9
10    if i < len(my_list):
11        # Found an item with the property
12        return True
13    else:
14        # There is no item with the property
15        return False

```

This could also be done with a `for` loop. In this case, the loop will exit early by using a `return` once the item has been found. The code is shorter, but not every programmer would prefer it. Some programmers feel that loops should not be prematurely ended with a `return` or `break` statement. It all goes to personal preference, or the personal preference of the person that is footing the bill.

Listing 125: Check if list has an item that has a property - for loop

```

1 def check_if_one_item_is_in_room_v2(my_list, room):
2     """
3         Return true if at least one item has a
4         property. Works the same as v1, but less code.
5     """
6     for item in my_list:
7         if item.room == room:
8             return True
9     return False

```

Do All Items Have a Property?

Are all the adventure objects in the same room? This code is very similar to the prior example. Spot the difference and see if you can figure out the reason behind the change.

Listing 126: Check if all items have a property

```
1 def check_if_all_items_are_in_room(my_list, room):
2     """
3     Return true if at ALL items have a property.
4     """
5     for item in my_list:
6         if item.room != room:
7             return False
8     return True
```

Create a List With All Items Matching a Property

What if you wanted a list of objects that are in room 5? This is a combination of our prior code, and the code to append items to a list that we learned about back in [Introduction to Lists](#).

Listing 127: Create another list with all items matching a property

```
1 def get_items_in_room(my_list, room):
2     """
3     Build a brand new list that holds all the items
4     that match our property.
5     """
6     matching_list = []
7     for item in my_list:
8         if item.room == room:
9             matching_list.append(item)
10    return matching_list
```

How would you run all these in a test? The code above can be combined with this code to run:

Listing 128: Run Sample Functions

```
1 def main():
2     object_list = []
3     object_list.append(AdventureObject("Key", 5))
4     object_list.append(AdventureObject("Bear", 5))
5     object_list.append(AdventureObject("Food", 8))
6     object_list.append(AdventureObject("Sword", 2))
7     object_list.append(AdventureObject("Wand", 10))
8
9     result = check_if_one_item_has_property_v1(object_list, 5)
10    print("Result of test check_if_one_item_has_property_v1:", result)
11
12    result = check_if_one_item_has_property_v2(object_list, 5)
13    print("Result of test check_if_one_item_has_property_v2:", result)
14
15    result = check_if_all_items_have_property(object_list, 5)
16    print("Result of test check_if_all_items_have_property:", result)
17
18    result = get_matching_items(object_list, 5)
19    print("Number of items returned from test get_matching_items:", len(result))
20
21
22 main()
```

For a full working example:

Listing 129: linear_search_variations.py

```

1  class AdventureObject:
2      """ Class that defines an alien"""
3
4      def __init__(self, description, room):
5          """ Constructor. Set name and color"""
6          self.description = description
7          self.room = room
8
9
10     def check_if_one_item_is_in_room_v1(my_list, room):
11         """
12             Return true if at least one item has a
13             property.
14         """
15         i = 0
16         while i < len(my_list) and my_list[i].room != room:
17             i += 1
18
19         if i < len(my_list):
20             # Found an item with the property
21             return True
22         else:
23             # There is no item with the property
24             return False
25
26
27     def check_if_one_item_is_in_room_v2(my_list, room):
28         """
29             Return true if at least one item has a
30             property. Works the same as v1, but less code.
31         """
32         for item in my_list:
33             if item.room == room:
34                 return True
35         return False
36
37
38     def check_if_all_items_are_in_room(my_list, room):
39         """
40             Return true if at ALL items have a property.
41         """
42         for item in my_list:
43             if item.room != room:
44                 return False
45         return True
46
47
48     def get_items_in_room(my_list, room):
49         """
50             Build a brand new list that holds all the items
51             that match our property.
52         """
53         matching_list = []
54         for item in my_list:

```

(continues on next page)

(continued from previous page)

```

55     if item.room == room:
56         matching_list.append(item)
57
58
59
60 def main():
61     object_list = []
62     object_list.append(AdventureObject("Key", 5))
63     object_list.append(AdventureObject("Bear", 5))
64     object_list.append(AdventureObject("Food", 8))
65     object_list.append(AdventureObject("Sword", 2))
66     object_list.append(AdventureObject("Wand", 10))
67
68     result = check_if_one_item_is_in_room_v1(object_list, 5)
69     print("Result of test check_if_one_item_is_in_room_v1:", result)
70
71     result = check_if_one_item_is_in_room_v2(object_list, 5)
72     print("Result of test check_if_one_item_is_in_room_v2:", result)
73
74     result = check_if_all_items_are_in_room(object_list, 5)
75     print("Result of test check_if_all_items_are_in_room:", result)
76
77     result = get_items_in_room(object_list, 5)
78     print("Number of items returned from test get_items_in_room:", len(result))
79
80
81 main()

```

These common algorithms can be used as part of a solution to a larger problem, such as find all the addresses in a list of customers that aren't valid.

1.24.7 Binary Search

A faster way to search a list is possible with the binary search. The process of a binary search can be described by using the classic number guessing game “guess a number between 1 and 100” as an example. To make it easier to understand the process, let's modify the game to be “guess a number between 1 and 128.” The number range is inclusive, meaning both 1 and 128 are possibilities.

If a person were to use the linear search as a method to guess the secret number, the game would be rather long and boring.

```

Guess a number 1 to 128: 1
Too low.
Guess a number 1 to 128: 2
Too low.
Guess a number 1 to 128: 3
Too low.
....
Guess a number 1 to 128: 93
Too low.
Guess a number 1 to 128: 94
Correct!

```

Most people will use a binary search to find the number. Here is an example of playing the game using a binary search:

```
Guess a number 1 to 128: 64
Too low.
Guess a number 1 to 128: 96
Too high.
Guess a number 1 to 128: 80
Too low.
Guess a number 1 to 128: 88
Too low.
Guess a number 1 to 128: 92
Too low.
Guess a number 1 to 128: 94
Correct!
```

Each time through the rounds of the number guessing game, the guesser is able to eliminate one half of the problem space by getting a “high” or “low” as a result of the guess.

In a binary search, it is necessary to track an upper and a lower bound of the list that the answer can be in. The computer or number-guessing human picks the midpoint of those elements. Revisiting the example:

A lower bound of 1, upper bound of 128, mid point of $\frac{128+1}{2} = 64.5$.

```
Guess a number 1 to 128: 64
Too low.
```

A lower bound of 65, upper bound of 128, mid point of $\frac{65+128}{2} = 96.5$.

```
Guess a number 1 to 128: 96
Too high.
```

A lower bound of 65, upper bound of 95, mid point of $\frac{65+95}{2} = 80$.

```
Guess a number 1 to 128: 80
Too low.
```

A lower bound of 81, upper bound of 95, mid point of $\frac{81+95}{2} = 88$.

```
Guess a number 1 to 128: 88
Too low.
```

A lower bound of 89, upper bound of 95, mid point of $\frac{89+95}{2} = 92$.

```
Guess a number 1 to 128: 92
Too low.
```

A lower bound of 93, upper bound of 95, mid point of $\frac{93+95}{2} = 94$.

```
Guess a number 1 to 128: 94
Correct!
```

A binary search requires significantly fewer guesses. Worst case, it can guess a number between 1 and 128 in 7 guesses. One more guess raises the limit to 256. 9 guesses can get a number between 1 and 512. With just 32 guesses, a person can get a number between 1 and 4.2 billion.

To figure out how large the list can be given a certain number of guesses, the formula works out like $n = x^g$ where n is the size of the list and g is the number of guesses. For example:

$$2^7 = 128 \text{ (7 guesses can handle 128 different numbers)}$$

$$2^8 = 256$$

$$2^9 = 512$$

$$2^{32} = 4,294,967,296$$

If you have the problem size, we can figure out the number of guesses using the log function. Specifically, *log base 2*. If you don't specify a base, most people will assume you mean the natural log with a base of $e \approx 2.71828$ which is not what we want. For example, using log base 2 to find how many guesses:

$$\log_2 128 = 7$$

$$\log_2 65,536 = 16$$

Enough math! Where is the code? The code to do a binary search is more complex than a linear search:

Listing 130: Binary search

```
1 # --- Binary search
2 key = "Morgiana the Shrew"
3 lower_bound = 0
4 upper_bound = len(name_list)-1
5 found = False
6
7 # Loop until we find the item, or our upper/lower bounds meet
8 while lower_bound <= upper_bound and not found:
9
10    # Find the middle position
11    middle_pos = (lower_bound + upper_bound) // 2
12
13    # Figure out if we:
14    # move up the lower bound, or
15    # move down the upper bound, or
16    # we found what we are looking for
17    if name_list[middle_pos] < key:
18        lower_bound = middle_pos + 1
19    elif name_list[middle_pos] > key:
20        upper_bound = middle_pos - 1
21    else:
22        found = True
23
24 if found:
25     print( "The name is at position", middle_pos)
26 else:
27     print( "The name was not in the list." )
```

Since lists start at element zero, line 3 sets the lower bound to zero. Line 4 sets the upper bound to the length of the list minus one. So for a list of 100 elements the lower bound will be 0 and the upper bound 99.

The Boolean variable on line 5 will be used to let the while loop know that the element has been found.

Line 8 checks to see if the element has been found or if we've run out of elements. If we've run out of elements the lower bound will end up equaling the upper bound.

Line 11 finds the middle position. It is possible to get a middle position of something like 64.5. It isn't possible to look

up position 64.5. (Although J.K. Rowling was rather clever in enough coming up with Platform 9 $\frac{3}{4}$, that doesn't work here.) The best way of handling this is to use the `//` operator first introduced way back in Chapter 1. This is similar to the `/` operator, but will only return integer results. For example, `11 // 2` would give 5 as an answer, rather than 5.5.

Starting at line 17, the program checks to see if the guess is high, low, or correct. If the guess is low, the lower bound is moved up to just past the guess. If the guess is too high, the upper bound is moved just below the guess. If the answer has been found, `found` is set to `True` ending the search.

With the a list of 100 elements, a person can reasonably guess that on average with the linear search, a program will have to check 50 of them before finding the element. With the binary search, on average you'll still need to do about seven guesses. In an advanced algorithms course you can find the exact formula. For this course, just assume average and worst cases are the same.

1.25 Array-Backed Grids

1.25.1 Introduction

Games like minesweeper, tic-tac-toe, and many types of adventure games keep data for the game in a grid of numbers. For example, a tic-tac-toe board:

	O	O
X		
X		

... can use a grid of numbers to represent the empty spots, the O's and the X's like this:

0	2	2
0	1	0
1	0	0

This grid of numbers can also be called a *two-dimensional array* or a *matrix*. (Finally, we get to learn about The Matrix.) The values of the numbers in the grid represent what should be displayed at each board location. In the prior example, 0 represents a spot where no one has played, a 1 represents an X, and a 2 represents an O.

The figure above is an example from the classic minesweeper game. This example has been modified to show both the classic display on the left, and the grid of numbers used to display the board on the right.

The number 10 represents a mine, the number 0 represents a space that has not been clicked, and the number 9 represents a cleared space. The numbers 1 to 8 represent how many mines are within the surrounding eight squares, and is only filled in when the user clicks on the square.

Minesweeper can actually have two grids. One for the regular display, and a completely separate grid of numbers that will track if the user has placed “flags” on the board marking where she thinks the mines are.

Classic adventure game maps are created using a tiled map editor. These are huge grids where each location is simply a number representing the type of terrain that goes there. The terrain could be things like dirt, a road, a path, green grass, brown grass, and so forth. Programs like [Tiled](#) shown in the figure below allow a developer to easily make these maps and write the grid to disk.

Adventure games also use multiple grids of numbers, just like minesweeper has a grid for the mines, and a separate grid for the flags. One grid, or “layer,” in the adventure game represents terrain you can walk on; another for things you can’t walk on like walls and trees; a layer for things that can instantly kill you, like lava or bottomless pits; one for objects that can be picked up and moved around; and yet another layer for initial placement of monsters.

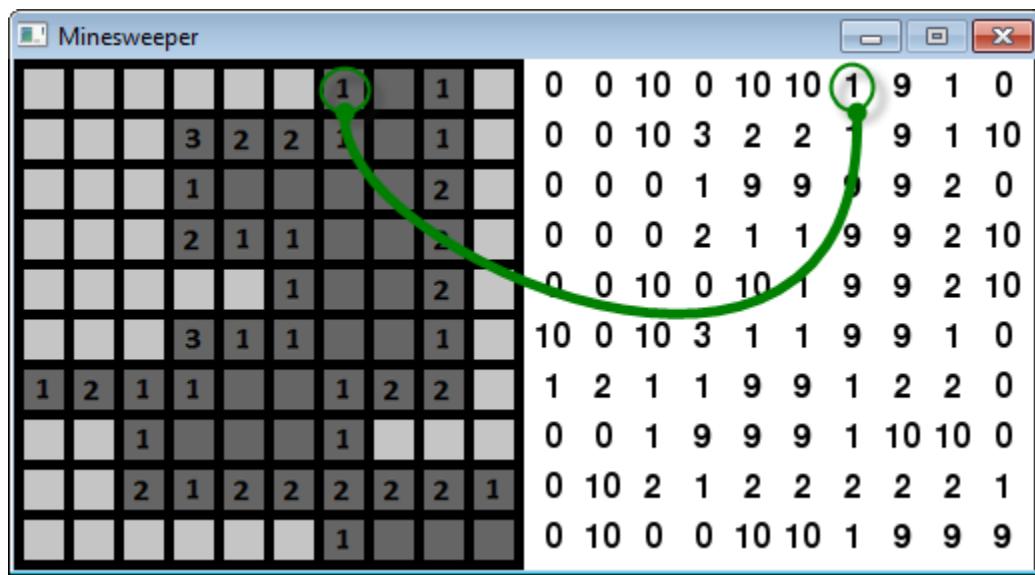


Fig. 54: Figure 16.1: Minesweeper game, showing the backing grid of numbers



Fig. 55: Figure 16.2: Using Qt Tiles to create an adventure map

Maps like these can be loaded in a Python program, but unfortunately a full description of how to manage is beyond the scope of this book. Projects like [PyTMX](#) that provide some of the code needed to load these maps.

1.25.2 Application

Enough talk, let's write some code. This example will create a grid that will trigger if we display a white or green block. We can change the grid value and make it green by clicking on it. This is a first step to a grid-based game like minesweeper, battleship, connect four, etc. (One year I had a student call me over and she had modified a program like this to show my name in flashing lights. That was ... disturbing. So please use this knowledge only for good!)

Start with this template:

Listing 131: array_backed_grid.py

```

1 import arcade
2
3 SCREEN_WIDTH = 500
4 SCREEN_HEIGHT = 600
5
6
7 class MyGame(arcade.Window):
8     """
9         Main application class.
10    """
11
12     def __init__(self, width, height):
13         super().__init__(width, height)
14
15         arcade.set_background_color(arcade.color.WHITE)
16
17     def on_draw(self):
18         """
19             Render the screen.
20         """
21
22         arcade.start_render()
23
24     def on_mouse_press(self, x, y, button, key_modifiers):
25         """
26             Called when the user presses a mouse button.
27         """
28         pass
29
30
31 def main():
32
33     window = MyGame(SCREEN_WIDTH, SCREEN_HEIGHT)
34     arcade.run()
35
36
37 if __name__ == "__main__":
38     main()
```

Starting with the file above, attempt to recreate this program following the instructions here. The final program is at the end of this chapter but don't skip ahead and copy it! If you do that you'll have learned nothing. Anyone can copy and paste the code, but if you can recreate this program you have skills people are willing to pay for. If you can only copy and paste, you've wasted your time here.

Drawing the Grid

1. Create variables named WIDTH, HEIGHT, and MARGIN. Set the width and height to 20. This will represent how large each grid location is. Set the margin to 5. This represents the margin between each grid location and the edges of the screen. Create these variables at the top of the program, after the `import` statements. Also create variables ROW_COUNT and COLUMN_COUNT. Set them to 10. This will control how many rows and columns we will have.
2. Calculate SCREEN_WIDTH and SCREEN_HEIGHT based on the variables we created above. If we have 10 rows, and each row is 20 high, that's 200 pixels. If we have 10 rows, that's also 11 margins. (Nine between the cells and two on each edge.) That is 55 more pixels for a total of 255. Write the equation so it works with whatever we select in the constants created by step 1.
3. Change the background to black. Draw a white box in the lower-left corner. Draw the box drawn using the height and width variables created earlier. (Feel free to adjust the colors.) Use the `draw_rectangle_filled` function. You will need to center the rectangle not at (0, 0) but at a coordinate that takes into account the height and width of the rectangle, such as $\frac{width}{2}$. When you get done your program's window should look like:

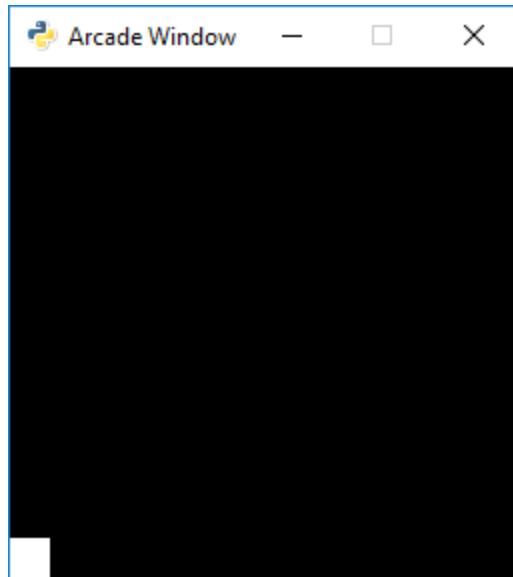


Fig. 56: Figure 16.3: Step 3

4. Use a `for` loop to draw COLUMN_COUNT boxes in a row. Use `column` for the variable name in the `for` loop. The output will look like one long box until we add in the margin between boxes. See Figure 16.4.
5. Adjust the drawing of the rectangle to add in the MARGIN variable. Now there should be gaps between the rectangles. See Figure 16.5.
6. Add the margin before drawing the rectangles, in addition to between each rectangle. This should keep the box from appearing right next to the window edge. See Figure 16.6. You'll end up with an equation like: $(margin + width) \cdot column + margin + \frac{width}{2}$
7. Add another `for` loop that also will loop for each row. Call the variable in this `for` loop `row`. Now we should have a full grid of boxes. See Figure 16.7.

Populating the Grid

8. Now we need to create a two-dimensional array. We need to create this once, at program start-up. So this will go in the program's `__init__` method. Creating a two-dimensional array in Python is, unfortunately, not as

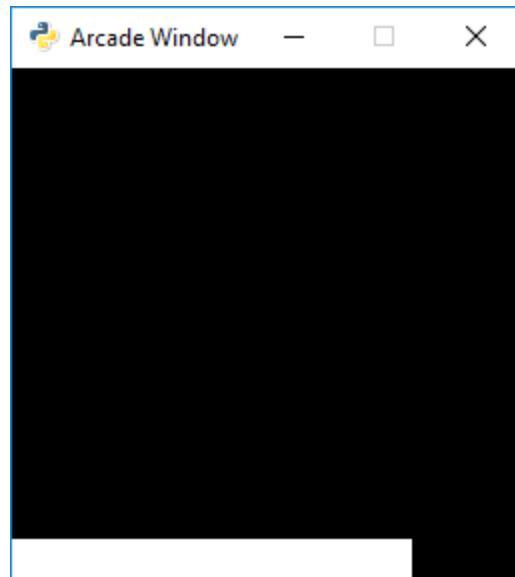


Fig. 57: Figure 16.4: Step 4

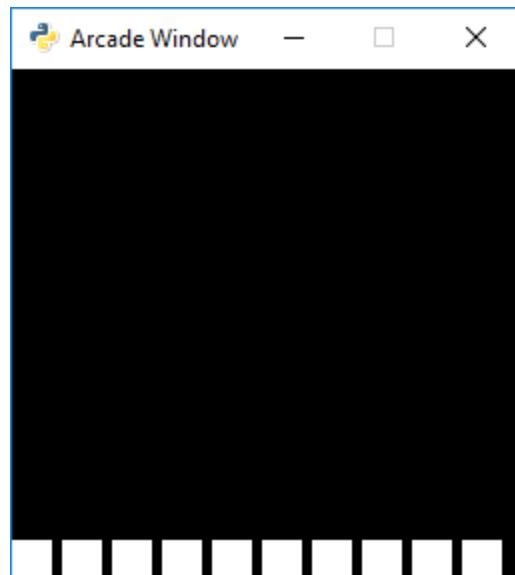


Fig. 58: Figure 16.5: Step 5

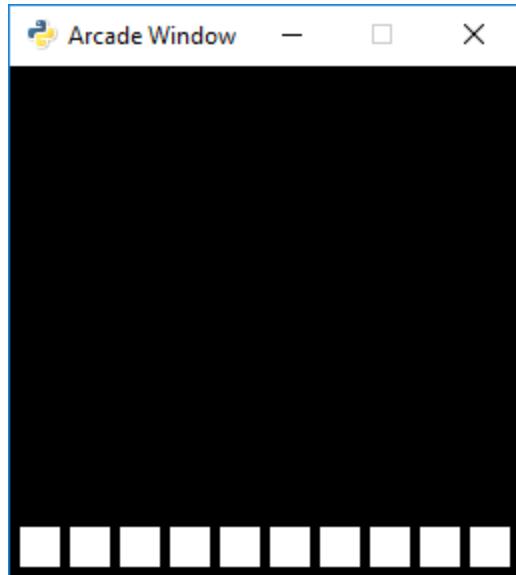


Fig. 59: Figure 16.6: Step 6

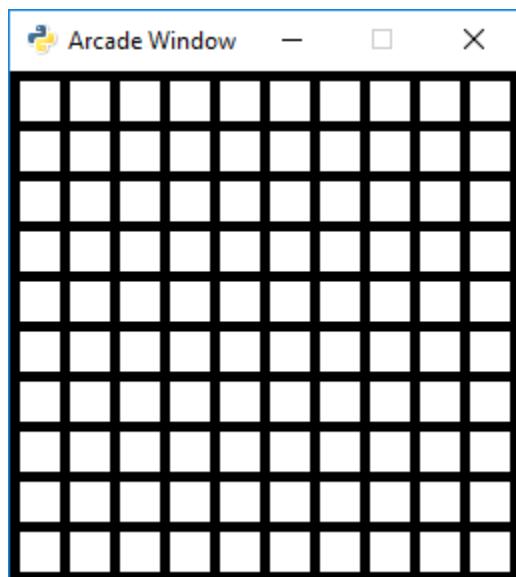


Fig. 60: Figure 16.7: Step 7

easy as it is in some other computer languages. There are some libraries that can be downloaded for Python that make it easy (like numpy), but for this example they will not be used. To create a two-dimensional array and set an example, use the code below:

Listing 132: Create a 10x10 array of numbers

```
ROW_COUNT = 10
COLUMN_COUNT = 10

# --- Create grid of numbers
# Create an empty list
self.grid = []
# Loop for each row
for row in range(ROW_COUNT):
    # For each row, create a list that will
    # represent an entire row
    self.grid.append([])
    # Loop for each column
    for column in range(COLUMN_COUNT):
        # Add a the number zero to the current row
        self.grid[row].append(0)
```

A much shorter example is below, but this example uses some odd parts of Python that I don't bother to explain in this book:

Listing 133: Create a 10x10 array of numbers

```
self.grid = [[0 for x in range(10)] for y in range(10)]
```

Use one of these two examples and place the code to create our array ahead of your main program loop.

9. Set an example location in the array to 1.

Two dimensional arrays are usually represented addressed by first their row, and then the column. This is called a row-major storage. Most languages use row-major storage, with the exception of Fortran and MATLAB. Fortran and MATLAB use column-major storage.

```
# Set row 1, column 5 to one
self.grid[1][5] = 1
```

Place this code somewhere ahead of your main program loop.

10. Select the color of the rectangle based on the value of a variable named `color`. Do this by first finding the line of code where the rectangle is drawn. Ahead of it, create a variable named `color` and set it equal to white. Then replace the white color in the rectangle declaration with the `color` variable.
11. Select the color based on the value in the grid. After setting `color` to white, place an if statement that looks at the value in `grid[row][column]` and changes the color to green if the grid value is equal to 1. There should now be one green square. See Figure 16.8.
12. Print “click” to the screen if the user clicks the mouse button. See [Mouse Clicks](#) if you’ve forgotten how to do that.
13. Print the mouse coordinates when the user clicks the mouse.
14. Convert the mouse coordinates into grid coordinates. Print those instead. Remember to use the width and height of each grid location combined with the margin. It will be necessary to convert the final value to an integer. This can be done by using `int` or by using the integer division operator `//` instead of the normal division operator `/`. See Figure 16.9.

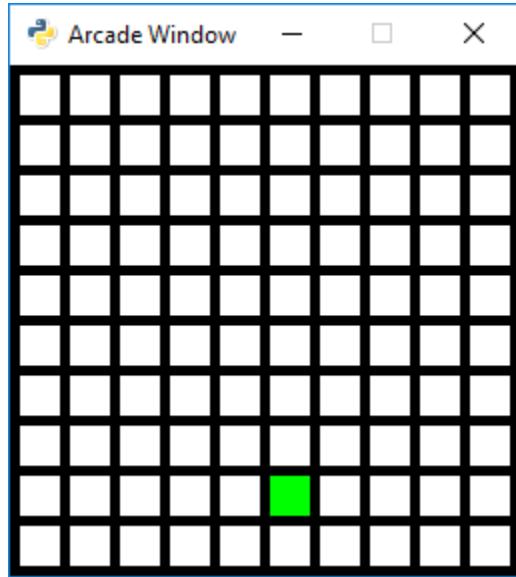


Fig. 61: Figure 16.8: Step 11

```
Click coordinates: (67, 142). Grid coordinates: (5, 2)
Click coordinates: (60, 109). Grid coordinates: (4, 2)
Click coordinates: (82, 96). Grid coordinates: (3, 3)
Click coordinates: (95, 139). Grid coordinates: (5, 3)
Click coordinates: (84, 160). Grid coordinates: (6, 3)
Click coordinates: (34, 143). Grid coordinates: (5, 1)
Click coordinates: (116, 105). Grid coordinates: (4, 4)
Click coordinates: (136, 100). Grid coordinates: (4, 5)
Click coordinates: (60, 36). Grid coordinates: (1, 2)
Click coordinates: (57, 40). Grid coordinates: (1, 2)
Click coordinates: (9, 23). Grid coordinates: (0, 0)
```

Fig. 62: Figure 16.9: Step 14

15. Set the grid location at the row/column clicked to 1. See Figure 16.10.

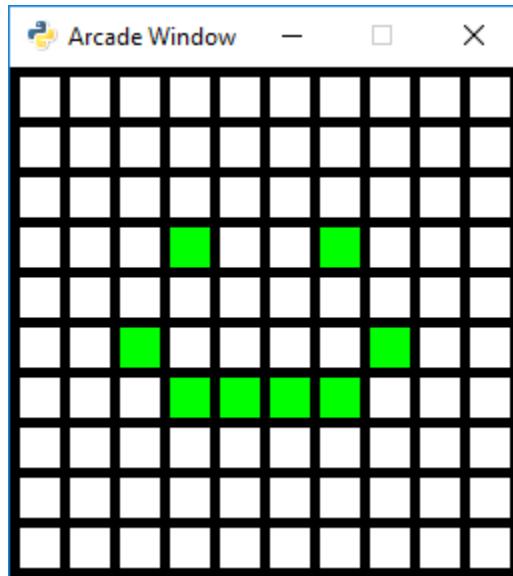


Fig. 63: Figure 16.10: Step 15

Final Program

Listing 134: array_backed_grid.py

```

1 """
2 Array Backed Grid
3
4 Show how to use a two-dimensional list/array to back the display of a
5 grid on-screen.
6 """
7 import arcade
8
9 # Set how many rows and columns we will have
10 ROW_COUNT = 10
11 COLUMN_COUNT = 10
12
13 # This sets the WIDTH and HEIGHT of each grid location
14 WIDTH = 20
15 HEIGHT = 20
16
17 # This sets the margin between each cell
18 # and on the edges of the screen.
19 MARGIN = 5
20
21 # Do the math to figure out our screen dimensions
22 SCREEN_WIDTH = (WIDTH + MARGIN) * COLUMN_COUNT + MARGIN
23 SCREEN_HEIGHT = (HEIGHT + MARGIN) * ROW_COUNT + MARGIN
24
25
26 class MyGame(arcade.Window):
27     """

```

(continues on next page)

(continued from previous page)

```

28     Main application class.
29     """
30
31     def __init__(self, width, height):
32         """
33             Set up the application.
34         """
35         super().__init__(width, height)
36         # Create a 2 dimensional array. A two dimensional
37         # array is simply a list of lists.
38         self.grid = []
39         for row in range(ROW_COUNT):
40             # Add an empty array that will hold each cell
41             # in this row
42             self.grid.append([])
43             for column in range(COLUMN_COUNT):
44                 self.grid[row].append(0)    # Append a cell
45
46         arcade.set_background_color(arcade.color.BLACK)
47
48     def on_draw(self):
49         """
50             Render the screen.
51         """
52
53         # This command has to happen before we start drawing
54         arcade.start_render()
55
56         # Draw the grid
57         for row in range(ROW_COUNT):
58             for column in range(COLUMN_COUNT):
59                 # Figure out what color to draw the box
60                 if self.grid[row][column] == 1:
61                     color = arcade.color.GREEN
62                 else:
63                     color = arcade.color.WHITE
64
65                 # Do the math to figure out where the box is
66                 x = (MARGIN + WIDTH) * column + MARGIN + WIDTH // 2
67                 y = (MARGIN + HEIGHT) * row + MARGIN + HEIGHT // 2
68
69                 # Draw the box
70                 arcade.draw_rectangle_filled(x, y, WIDTH, HEIGHT, color)
71
72     def on_mouse_press(self, x, y, button, modifiers):
73         """
74             Called when the user presses a mouse button.
75         """
76
77         # Change the x/y screen coordinates to grid coordinates
78         column = x // (WIDTH + MARGIN)
79         row = y // (HEIGHT + MARGIN)
80
81         print(f"Click coordinates: ({x}, {y}). Grid coordinates: ({row}, {column})")
82
83         # Make sure we are on-grid. It is possible to click in the upper right
84         # corner in the margin and go to a grid location that doesn't exist

```

(continues on next page)

(continued from previous page)

```

85     if row < ROW_COUNT and column < COLUMN_COUNT:
86
87         # Flip the location between 1 and 0.
88         if self.grid[row][column] == 0:
89             self.grid[row][column] = 1
90         else:
91             self.grid[row][column] = 0
92
93
94     def main():
95
96         window = MyGame(SCREEN_WIDTH, SCREEN_HEIGHT)
97         arcade.run()
98
99
100    if __name__ == "__main__":
101        main()

```

1.26 Platformers

Ever wanted to create your own platformer? It isn't too hard! Here's an example to get started.

1.26.1 Map File

Creating The Map

First, we need a map. This is a “map” file created with the [Tiled](#) program. The program is free. You can download it and use it to create your map file.

In this map file the numbers represent:

Number	Item
-1	Empty square
0	Crate
1	Left grass corner
2	Middle grass corner
3	Right grass corner

You can download these tiles (originally from kenney.nl) here:





Of course, you'll need a character to jump around the map:



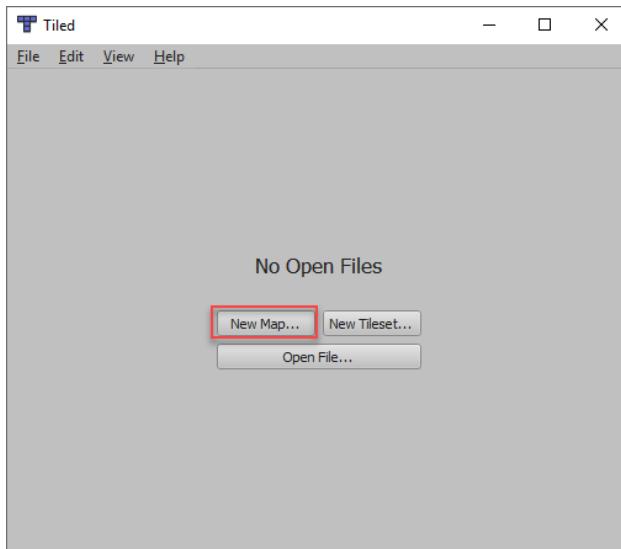
Here is the map file:

Listing 135: map.csv

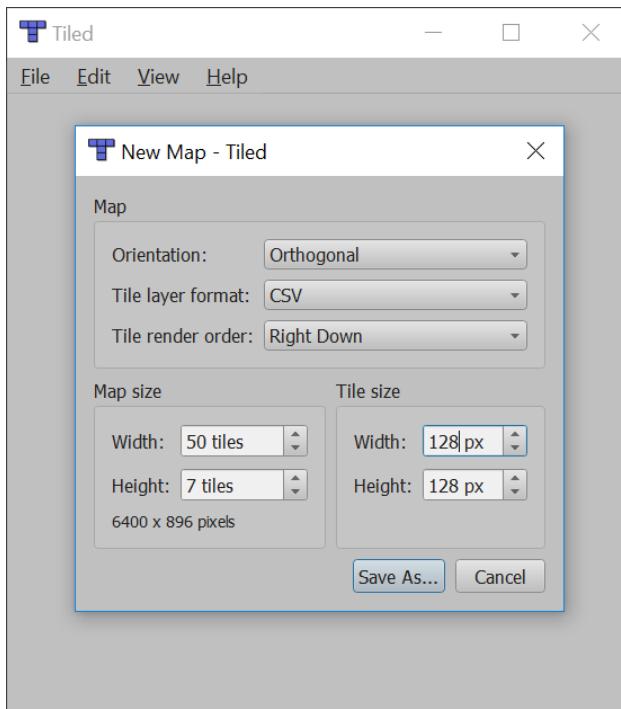
(continues on next page)

(continued from previous page)

The **Tiled** program takes some getting used to. You start off with a screen like this, and you can create a new map here:

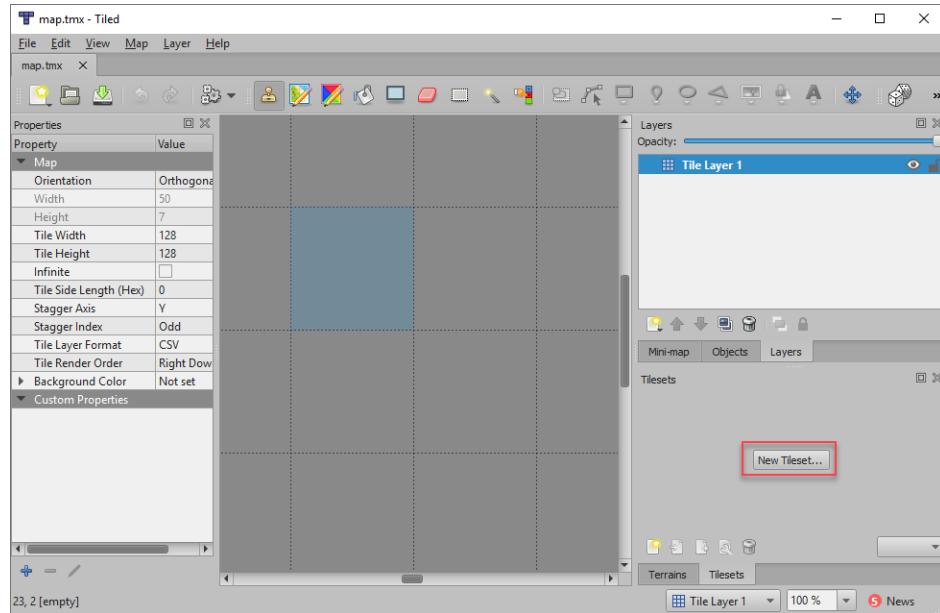


Then set up your map like this, adjusting the size of the map and the size of your images accordingly. (All your images need to be the same size. Don't guess this number, look at the properties of the image and find how big it is.)

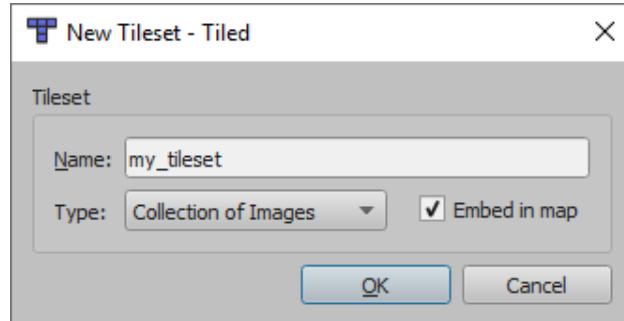


Most of the tiles from kenney.nl are 128x128 pixels. In the image above I've got a 7 tile high, by 50 pixel wide side-scroll map.

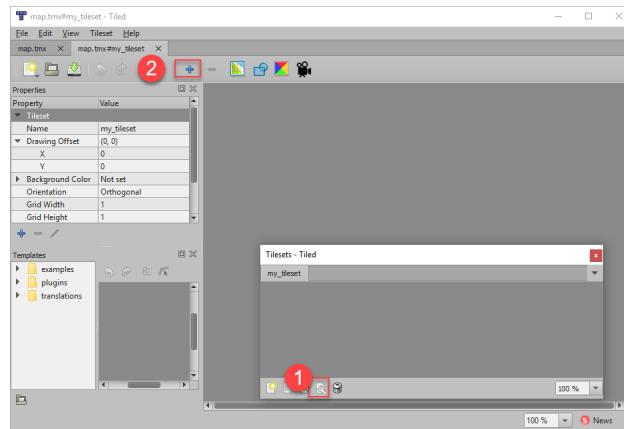
After this, you have to create a new “tile set.” Find the button for that:



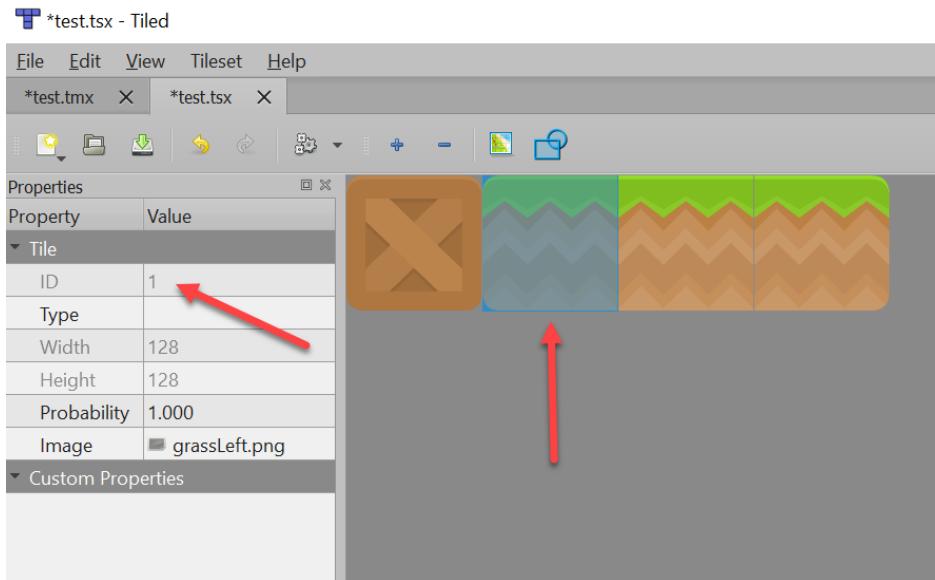
I use these settings:



You can add the images as tiles to your tileset. I don't find this obvious, but you click the wrench icon, then the plus icon:

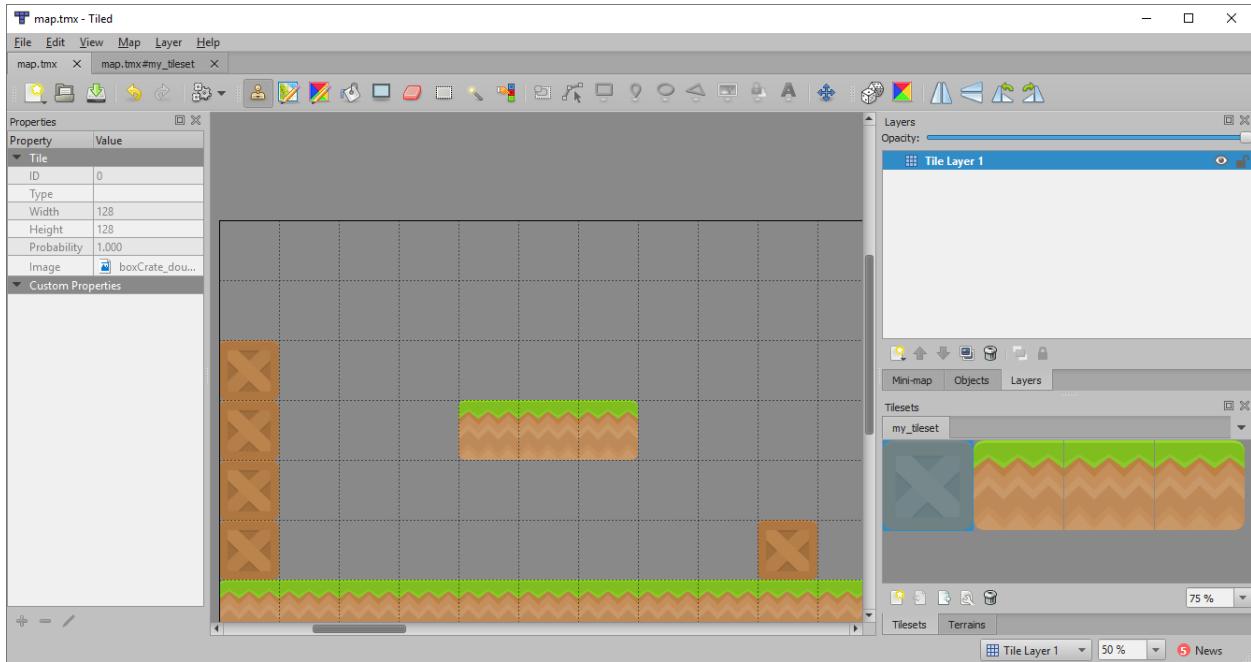


These “tiles” will be all the images for your map, and the numbers they associate with:



The numbers of the tiles correspond to the order you added the tiles. I don't think you can change the mapping after you create the tileset.

Next, you "paint" your map:



When you are done, you can "Export as" a CSV file.

Reading The Map

Next, we want to read in this grid of numbers where each number is separated by a comma. We know how to read in a file, but how do you process a comma delimited file?

We've learned how to take a string and use the functions:

- `upper()`

- `lower()`
- `strip()`

There's another function called `split()`. This function will split apart a string into a list based on a delimiter. For example:

```
1 # A comma delimited string of numbers
2 my_text_string = "23,34,1,3,5,10"
3
4 # Split the string into a list, based on the comma as a delimiter
5 my_list = my_text_string.split(",")
6
7 # Print the result
8 print(my_list)
```

This prints:

```
['23', '34', '1', '3', '5', '10']
```

Which is close to what we want, except the list is a list of text, not numbers.

We can convert the list by:

```
# Convert from list of strings to list of integers
for i in range(len(my_list)):
    my_list[i] = int(my_list[i])
```

We haven't covered it a lot, but you can also use `enumerate` to do the same thing:

```
# Convert from list of strings to list of integers
for index, item in enumerate(my_list):
    my_list[index] = int(item)
```

Or use a list comprehension:

```
# Convert from list of strings to list of integers
my_list = [int(item) for item in my_list]
```

Python does have built-in code for working with csv files. If you want, you can read about the *csv library* in the official documentation.

1.26.2 Platformer Physics Engine

In prior chapters, we've used the `PhysicsEngineSimple` to keep from running through walls. There's another engine called `PhysicsEnginePlatformer` for platformers.

This engine has two important additions:

1. Gravity
2. `can_jump` method

Gravity

Creating the platformer physics engine requires a gravity constant. I recommend 0.5 to start with. This is your acceleration in pixels per frame.

```
self.physics_engine = arcade.PhysicsEnginePlatformer(self.player_sprite,
                                                    self.wall_list,
                                                    gravity_constant=GRAVITY)
```

Jumping

Also, you often need to know if there is ground beneath your character to know if she can jump. The physics engine has a method for this:

```
if self.physics_engine.can_jump():
    self.player_sprite.change_y = JUMP_SPEED
```

1.26.3 Python Program

In the highlighted code for the listing below, see how we've implemented these concepts to create a platformer

Listing 136: Platformer example, simple

```
"""
Load a map stored in csv format, as exported by the program 'Tiled.'

Artwork from: http://kenney.nl
Tiled available from: http://www.mapeditor.org/
"""

import arcade

SPRITE_SCALING = 0.5

SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600

# How many pixels to keep as a minimum margin between the character
# and the edge of the screen.
VIEWPORT_MARGIN = 40
RIGHT_MARGIN = 150

TILE_SIZE = 128
SCALED_TILE_SIZE = TILE_SIZE * SPRITE_SCALING
MAP_HEIGHT = 7

# Physics
MOVEMENT_SPEED = 5
JUMP_SPEED = 14
GRAVITY = 0.5

def get_map(filename):
    """
    This function loads an array based on a map stored as a list of
    numbers separated by commas.
    """

    # Open the file
    map_file = open(filename)
```

(continues on next page)

(continued from previous page)

```

37     # Create an empty list of rows that will hold our map
38     map_array = []
39
40     # Read in a line from the file
41     for line in map_file:
42
43         # Strip the whitespace, and \n at the end
44         line = line.strip()
45
46         # This creates a list by splitting line everywhere there is a comma.
47         map_row = line.split(",")
48
49         # The list currently has all the numbers stored as text, and we want it
50         # as a number. (e.g. We want 1 not "1"). So loop through and convert
51         # to an integer.
52         for index, item in enumerate(map_row):
53             map_row[index] = int(item)
54
55         # Now that we've completed processing the row, add it to our map array.
56         map_array.append(map_row)
57
58     # Done, return the map.
59     return map_array
60
61
62
63 class MyWindow(arcade.Window):
64     """ Main application class. """
65
66     def __init__(self):
67         """
68         Initializer
69         """
70
71         # Call the parent class
72         super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT)
73
74         # Sprite lists
75         self.player_list = None
76         self.wall_list = None
77
78         # Set up the player
79         self.player_sprite = None
80
81         # Physics engine
82         self.physics_engine = None
83
84         # Used for scrolling map
85         self.view_left = 0
86         self.view_bottom = 0
87
88     def setup(self):
89         """
90         Set up the game and initialize the variables.
91         """
92
93         # Sprite lists
94         self.player_list = arcade.SpriteList()
95         self.wall_list = arcade.SpriteList()

```

(continues on next page)

(continued from previous page)

```

94     # Set up the player
95     self.player_sprite = arcade.Sprite("character.png", SPRITE_SCALING)

96
97     # Starting position of the player
98     self.player_sprite.center_x = 90
99     self.player_sprite.center_y = 270
100    self.player_list.append(self.player_sprite)

101
102    # Get a 2D array made of numbers based on the map
103    map_array = get_map("map.csv")

104
105    # Now that we've got the map, loop through and create the sprites
106    for row_index in range(len(map_array)):
107        for column_index in range(len(map_array[row_index])):

108            item = map_array[row_index][column_index]

109
110            # For this map, the numbers represent:
111            # -1 = empty
112            # 0 = box
113            # 1 = grass left edge
114            # 2 = grass middle
115            # 3 = grass right edge
116            if item == 0:
117                wall = arcade.Sprite("boxCrate_double.png", SPRITE_SCALING)
118            elif item == 1:
119                wall = arcade.Sprite("grassLeft.png", SPRITE_SCALING)
120            elif item == 2:
121                wall = arcade.Sprite("grassMid.png", SPRITE_SCALING)
122            elif item == 3:
123                wall = arcade.Sprite("grassRight.png", SPRITE_SCALING)

124
125            if item >= 0:
126                # Calculate where the sprite goes
127                wall.left = column_index * SCALED_TILE_SIZE
128                wall.top = (MAP_HEIGHT - row_index) * SCALED_TILE_SIZE

129
130            # Add the sprite
131            self.wall_list.append(wall)

132
133
134    # Create our platformer physics engine with gravity
135    self.physics_engine = arcade.PhysicsEnginePlatformer(self.player_sprite,
136                                                       self.wall_list,
137                                                       gravity_constant=GRAVITY)

138
139    # Set the background color
140    arcade.set_background_color(arcade.color.AMAZON)

141
142    # Set the view port boundaries
143    # These numbers set where we have 'scrolled' to.
144    self.view_left = 0
145    self.view_bottom = 0

146
147    def on_draw(self):
148        """
149        Render the screen.
150        """

```

(continues on next page)

(continued from previous page)

```

151
152     # This command has to happen before we start drawing
153     arcade.start_render()
154
155     # Draw all the sprites.
156     self.wall_list.draw()
157     self.player_list.draw()
158
159     def on_key_press(self, key, modifiers):
160         """
161             Called whenever the mouse moves.
162         """
163
164         if key == arcade.key.UP:
165             # This line below is new. It checks to make sure there is a platform
166             # underneath
167             # the player. Because you can't jump if there isn't ground beneath your
168             # feet.
169             if self.physics_engine.can_jump():
170                 self.player_sprite.change_y = JUMP_SPEED
171
172             elif key == arcade.key.LEFT:
173                 self.player_sprite.change_x = -MOVEMENT_SPEED
174             elif key == arcade.key.RIGHT:
175                 self.player_sprite.change_x = MOVEMENT_SPEED
176
177     def on_key_release(self, key, modifiers):
178         """
179             Called when the user presses a mouse button.
180         """
181
182         if key == arcade.key.LEFT or key == arcade.key.RIGHT:
183             self.player_sprite.change_x = 0
184
185     def update(self, delta_time):
186         """
187             Movement and game logic
188         """
189
190         self.physics_engine.update()
191
192         # --- Manage Scrolling ---
193
194         # Track if we need to change the view port
195
196         changed = False
197
198         # Scroll left
199         left_bndry = self.view_left + VIEWPORT_MARGIN
200         if self.player_sprite.left < left_bndry:
201             self.view_left -= left_bndry - self.player_sprite.left
202             changed = True
203
204         # Scroll right
205         right_bndry = self.view_left + SCREEN_WIDTH - RIGHT_MARGIN
206         if self.player_sprite.right > right_bndry:
207             self.view_left += self.player_sprite.right - right_bndry
208             changed = True
209
210         # Scroll up
211         top_bndry = self.view_bottom + SCREEN_HEIGHT - VIEWPORT_MARGIN
212         if self.player_sprite.top > top_bndry:

```

(continues on next page)

(continued from previous page)

```

206     self.view_bottom += self.player_sprite.top - top_bndry
207     changed = True
208
209     # Scroll down
210     bottom_bndry = self.view_bottom + VIEWPORT_MARGIN
211     if self.player_sprite.bottom < bottom_bndry:
212         self.view_bottom -= bottom_bndry - self.player_sprite.bottom
213         changed = True
214
215     # If we need to scroll, go ahead and do it.
216     if changed:
217         arcade.set_viewport(self.view_left,
218                             SCREEN_WIDTH + self.view_left,
219                             self.view_bottom,
220                             SCREEN_HEIGHT + self.view_bottom)
221
222
223 def main():
224     window = MyWindow()
225     window.setup()
226
227     arcade.run()
228
229
230 main()

```

1.26.4 Other Examples

- If you are looking for platforms that move, see [Sprite Moving Platforms](#).
- If you are looking to be able to create ramps you can run up and down, see [Sprite Ramps](#).

1.27 Sorting

Binary searches only work on lists that are in order. So how do programs get a list in order? How does a program sort a list of items when the user clicks a column heading, or otherwise needs something sorted?

There are several algorithms that do this. The two easiest algorithms for sorting are the *selection sort* and the *insertion sort*. Other sorting algorithms exist as well, such as the shell, merge, heap, and quick sorts.

The best way to get an idea on how these sorts work is to watch them. To see common sorting algorithms in action visit this excellent website:

<http://www.sorting-algorithms.com>

Each sort has advantages and disadvantages. Some sort a list quickly if the list is almost in order to begin with. Some sort a list quickly if the list is in a completely random order. Other lists sort fast, but take more memory. Understanding how sorts work is important in selecting the proper sort for your program.

1.27.1 Swapping Values

Before learning to sort, we need to learn how to swap values between two variables. This is a common operation in many sorting algorithms. Suppose a program has a list that looks like the following:

```
my_list = [15, 57, 14, 33, 72, 79, 26, 56, 42, 40]
```

The developer wants to swap positions 0 and 2, which contain the numbers 15 and 14 respectively. See Figure 18.1.

Fig. 64: Figure 27.1: Swapping values in an array

A first attempt at writing this code might look something like this:

```
my_list[0] = my_list[2]
my_list[2] = my_list[0]
```

Fig. 65: Figure 27.2: Incorrect attempt to swap array values

See Figure 27.2 to get an idea on what would happen. This clearly does not work. The first assignment `list[0] = list[2]` causes the value 15 that exists in position 0 to be overwritten with the 14 in position 2 and irretrievably lost. The next line with `list[2] = list[0]` just copies the 14 back to cell 2 which already has a 14.

To fix this problem, swapping values in an array should be done in three steps. It is necessary to create a temporary variable to hold a value during the swap operation. See Figure 18.3. The code to do the swap looks like the following:

Listing 137: Swapping two values in an array

```
temp = my_list[0]
my_list[0] = my_list[2]
my_list[2] = temp
```

The first line copies the value of position 0 into the `temp` variable. This allows the code to write over position 0 with the value in position 2 without data being lost. The final line takes the old value of position 0, currently held in the `temp` variable, and places it in position 2.

Fig. 66: Figure 27.3: Correct method to swap array values

1.27.2 Selection Sort

The selection by looking at element 0. Then code next scans the rest of the list from element 1 to n-1 to find the smallest number. The smallest number is swapped into element 0. The code then moves on to element 1, then 2, and so forth. Graphically, the sort looks like Figure 18.4.

The code for a selection sort involves two nested loops. The outside loop tracks the current position that the code wants to swap the smallest value into. The inside loop starts at the current location and scans to the right in search of the smallest value. When it finds the smallest value, the swap takes place.

Listing 138: Selection Sort

```
1 def selection_sort(my_list):
2     """ Sort a list using the selection sort """
3
4     # Loop through the entire array
5     for cur_pos in range(len(my_list)):
6         # Find the position that has the smallest number
7         # Start with the current position
```

(continues on next page)

Fig. 67: Figure 27.4: Selection Sort

(continued from previous page)

```

8     min_pos = cur_pos
9
10    # Scan left to right (end of the list)
11    for scan_pos in range(cur_pos + 1, len(my_list)):
12
13        # Is this position smallest?
14        if my_list[scan_pos] < my_list[min_pos]:
15
16            # It is, mark this position as the smallest
17            min_pos = scan_pos
18
19        # Swap the two values
20        temp = my_list[min_pos]
21        my_list[min_pos] = my_list[cur_pos]
22        my_list[cur_pos] = temp

```

The outside loop will always run n times. The inside loop will run an average of $\frac{n}{2}$ times per run of the outside loop. Therefore the inside loop will run a total of $n \cdot \frac{n}{2}$ or $\frac{n^2}{2}$ times.

This will be the case regardless if the list is in order or not. The loops' efficiency may be improved by checking if `min_pos` and `cur_pos` are equal before line 20. If those variables are equal, there is no need to do the three lines of swap code.

In order to test the selection sort code above, the following code may be used. The first function will print out the list. The next code will create a list of random numbers, print it, sort it, and then print it again. On line 5 the print statement right-aligns the numbers to make the column of numbers easier to read. Formatting `print` statements will be covered in a later chapter.

Listing 139: Code to create and print list to sort

```

# Before this code, paste the selection sort and import random
# Create a list of random numbers
my_list = []
for i in range(10):
    my_list.append(random.randrange(100))
# Try out the sort
print_list(my_list)
selection_sort(my_list)
print_list(my_list)

```

See an animation of the selection sort at:

<http://www.sorting-algorithms.com/selection-sort>

For a truly unique visualization of the selection sort, search YouTube for “selection sort dance” or use this link:

<http://youtu.be/Ns4TPTC8whw>

You also can trace through the code using [Selection Sort](#) on Python Tutor.

1.27.3 Insertion Sort

The insertion sort is similar to the selection sort in how the outer loop works. The insertion sort starts at the left side of the array and works to the right side. The difference is that the insertion sort does not select the smallest element and put it into place; the insertion sort selects the next element to the right of what was already sorted. Then it slides up each larger element until it gets to the correct location to insert. Graphically, it looks like Figure 18.5.

Fig. 68: Figure 27.5: Insertion Sort

The insertion sort breaks the list into two sections, the “sorted” half and the “unsorted” half. In each round of the outside loop, the algorithm will grab the next unsorted element and insert it into the list.

In the code below, the `key_pos` marks the boundary between the sorted and unsorted portions of the list. The algorithm scans to the left of `key_pos` using the variable `scan_pos`. Note that in the insertion sort, `scan_pos` goes down to the left, rather than up to the right. Each cell location that is larger than `key_value` gets moved up (to the right) one location.

When the loop finds a location smaller than `key_value`, it stops and puts `key_value` to the left of it.

The outside loop with an insertion sort will run n times. For each run of the outside loop, the inside loop will run an average of $\frac{n}{4}$ times if the loop is randomly shuffled. In total, the inside loop would run $n \cdot \frac{n}{4}$ times, or simplified, $\frac{n^2}{4}$ times.

What’s really important: If the loop is close to a sorted loop already, then the inside loop does not run very much, and the sort time is closer to n . The insertion sort is the fastest sort for nearly-sorted lists. If the list is reversed, then the insertion sort is terrible.

The selection sort doesn’t really care what order the list is in to begin with. It performs the same regardless.

Listing 140: Insertion Sort

```
1 def insertion_sort(my_list):
2     """ Sort a list using the insertion sort """
3
4     # Start at the second element (pos 1).
5     # Use this element to insert into the
6     # list.
7     for key_pos in range(1, len(my_list)):
8
9         # Get the value of the element to insert
10        key_value = my_list[key_pos]
11
12        # Scan from right to the left (start of list)
13        scan_pos = key_pos - 1
14
15        # Loop each element, moving them up until
16        # we reach the position the
17        while (scan_pos >= 0) and (my_list[scan_pos] > key_value):
18            my_list[scan_pos + 1] = my_list[scan_pos]
19            scan_pos = scan_pos - 1
20
21        # Everything's been moved out of the way, insert
22        # the key into the correct location
23        my_list[scan_pos + 1] = key_value
```

See an animation of the insertion sort at:

<http://www.sorting-algorithms.com/insertion-sort>

For another dance interpretation, search YouTube for “insertion sort dance” or use this link:

<http://youtu.be/ROalU379l3U>

You can trace through the code using [Insertion Sort](#) on Python Tutor.

1.27.4 Full Sorting Example

Listing 141: Full Sorting Example

```

1 import random
2
3
4 def selection_sort(my_list):
5     """ Sort a list using the selection sort """
6
7     # Loop through the entire array
8     for cur_pos in range(len(my_list)):
9         # Find the position that has the smallest number
10        # Start with the current position
11        min_pos = cur_pos
12
13        # Scan left to right (end of the list)
14        for scan_pos in range(cur_pos + 1, len(my_list)):
15
16            # Is this position smallest?
17            if my_list[scan_pos] < my_list[min_pos]:
18                # It is, mark this position as the smallest
19                min_pos = scan_pos
20
21        # Swap the two values
22        temp = my_list[min_pos]
23        my_list[min_pos] = my_list[cur_pos]
24        my_list[cur_pos] = temp
25
26
27 def insertion_sort(my_list):
28     """ Sort a list using the insertion sort """
29
30     # Start at the second element (pos 1).
31     # Use this element to insert into the
32     # list.
33     for key_pos in range(1, len(my_list)):
34
35         # Get the value of the element to insert
36         key_value = my_list[key_pos]
37
38         # Scan from right to the left (start of list)
39         scan_pos = key_pos - 1
40
41         # Loop each element, moving them up until
42         # we reach the position the
43         while (scan_pos >= 0) and (my_list[scan_pos] > key_value):
44             my_list[scan_pos + 1] = my_list[scan_pos]
```

(continues on next page)

(continued from previous page)

```
45     scan_pos = scan_pos - 1
46
47     # Everything's been moved out of the way, insert
48     # the key into the correct location
49     my_list[scan_pos + 1] = key_value
50
51
52 # This will point out a list
53 # For more information on the print formatting {:3}
54 # see the chapter on print formatting.
55 def print_list(my_list):
56     for item in my_list:
57         print(f"{item:3}", end="")
58     print()
59
60
61 def main():
62     # Create two lists of the same random numbers
63     list_for_selection_sort = []
64     list_for_insertion_sort = []
65     list_size = 10
66     for i in range(list_size):
67         new_number = random.randrange(100)
68         list_for_selection_sort.append(new_number)
69         list_for_insertion_sort.append(new_number)
70
71     # Print the original list
72     print("Original List")
73     print_list(list_for_selection_sort)
74
75     # Use the selection sort and print the result
76     print("Selection Sort")
77     selection_sort(list_for_selection_sort)
78     print_list(list_for_selection_sort)
79
80     # Use the insertion sort and print the result
81     print("Insertion Sort")
82     insertion_sort(list_for_insertion_sort)
83     print_list(list_for_insertion_sort)
84
85
86 main()
```

1.28 Exceptions

When something goes wrong with your program, do you want to keep the user from seeing a red Python error message? Do you want to keep your program from hanging? If so, then you need *exceptions*.

Exceptions are used to handle abnormal conditions that can occur during the execution of code. Exceptions are often used with file and network operations. This allows code to gracefully handle running out of disk space, network errors, or permission errors.

1.28.1 Vocabulary

There are several terms and phrases used while working with exceptions. Here are the most common:

- **Exception:** This term could mean one of two things. First, the condition that results in abnormal program flow. Or it could be used to refer to an object that represents the data condition. Each exception has an object that holds information about it.
- **Exception handling:** The process of handling an exception to normal program flow.
- **Catch block or exception block:** Code that handles an abnormal condition is said to “catch” the exception.
- **Throw or raise:** When an abnormal condition to the program flow has been detected, an instance of an exception object is created. It is then “thrown” or “raised” to code that will catch it.
- **Unhandled exception or Uncaught exception:** An exception that is thrown, but never caught. This usually results in an error and the program ending or crashing.
- **Try block:** A set of code that might have an exception thrown in it.

Most programming languages use the terms “throw” and “catch.” Unfortunately Python doesn’t. Python uses “raise” and “exception.” We introduce the throw/catch vocabulary here because they are the most prevalent terms in the industry.

1.28.2 Exception Handling

The code for handling exceptions is simple. See the example below:

Listing 142: Handling division by zero

```

1 # Divide by zero
2 try:
3     x = 5 / 0
4 except:
5     print("Error dividing by zero")

```

On line two is the `try` statement. Every indented line below it is part of the “try block.” There may be no unindented code below the `try` block that doesn’t start with an `except` statement. The `try` statement defines a section of code that the code will attempt to execute.

If there is any exception that occurs during the processing of the code the execution will immediately jump to the “catch block.” That block of code is indented under the `except` statement on line 4. This code is responsible for handling the error.

A program may use exceptions to catch errors that occur during a conversion from text to a number. For example:

Listing 143: Handling number conversion errors

```

1 # Invalid number conversion
2 try:
3     x = int("fred")
4 except:
5     print("Error converting fred to a number")

```

An exception will be thrown on line 3 because “fred” can not be converted to an integer. The code on line 5 will print out an error message.

Below is an expanded version on this example. It error-checks a user’s input to make sure an integer is entered. If the user doesn’t enter an integer, the program will keep asking for one. The code uses exception handling to capture a possible conversion error that can occur on line 5. If the user enters something other than an integer, an exception is

thrown when the conversion to a number occurs on line 5. The code on line 6 that sets `number_entered` to `True` will not be run if there is an exception on line 5.

Listing 144: Better handling of number conversion errors

```
1 number_entered = False
2 while not number_entered:
3     number_string = input("Enter an integer: ")
4     try:
5         n = int(number_string)
6         number_entered = True
7     except:
8         print("Error, invalid integer")
```

Files are particularly prone to errors during operations with them. A disk could fill up, a user could delete a file while it is being written, it could be moved, or a USB drive could be pulled out mid-operation. These types of errors may also be easily captured by using exception handling.

Listing 145: Checking for an error when opening a file

```
1 # Error opening file
2 try:
3     my_file = open("myfile.txt")
4 except:
5     print("Error opening file")
```

Multiple types of errors may be captured and processed differently. It can be useful to provide a more exact error message to the user than a simple “an error has occurred.”

In the code below, different types of errors can occur from lines 3-6. By placing `IOError` after `except` on line 7, only errors regarding Input and Output (IO) will be handled by that code. Likewise line 9 only handles errors around converting values, and line 11 covers division by zero errors. The last exception handling occurs on line 13. Since line 13 does not include a particular type of error, it will handle any error not covered by the `except` blocks above. The “catch-all” `except` must always be last.

Listing 146: Handling different types of errors

```
1 # Multiple errors
2 try:
3     # Open the file
4     filename = "myfile.txt"
5     my_file = open(filename)
6
7     # Read from the file and strip any trailing line feeds
8     my_line = my_file.readline()
9     my_line = my_line.strip()
10
11    # Convert to a number
12    my_int = int(my_line)
13
14    # Do a calculation
15    my_calculated_value = 101 / my_int
16
17 except FileNotFoundError:
18     print(f"Could not find the file '{filename}'")
19 except IOError:
20     print(f"Input/Output error when accessing the file '{filename}'")
21 except ValueError:
```

(continues on next page)

(continued from previous page)

```

22     print("Could not convert data to an integer.")
23 except ZeroDivisionError:
24     print("Division by zero error.")
25 except:
26     print("Unexpected error.")

```

A list of built-in exceptions is available from this web address:

<http://docs.python.org/library/exceptions.html>

1.28.3 Example: Saving High Score

This shows how to save a high score between games. The score is stored in a file called `high_score.txt`.

Listing 147: `high_score.py`

```

1 """
2 Show how to use exceptions to save a high score for a game.
3
4 Sample Python/Pygame Programs
5 Simpson College Computer Science
6 http://simpson.edu/computer-science/
7 """
8
9
10 def get_high_score():
11     # Default high score
12     high_score = 0
13
14     # Try to read the high score from a file
15     try:
16         high_score_file = open("high_score.txt", "r")
17         high_score = int(high_score_file.read())
18         high_score_file.close()
19         print("The high score is", high_score)
20     except IOError:
21         # Error reading file, no high score
22         print("There is no high score yet.")
23     except ValueError:
24         # There's a file there, but we don't understand the number.
25         print("I'm confused. Starting with no high score.")
26
27     return high_score
28
29
30 def save_high_score(new_high_score):
31     try:
32         # Write the file to disk
33         high_score_file = open("high_score.txt", "w")
34         high_score_file.write(str(new_high_score))
35         high_score_file.close()
36     except IOError:
37         # Hm, can't write it.
38         print("Unable to save the high score.")
39
40

```

(continues on next page)

(continued from previous page)

```

41 def main():
42     """ Main program is here. """
43     # Get the high score
44     high_score = get_high_score()
45
46     # Get the score from the current game
47     current_score = 0
48     try:
49         # Ask the user for his/her score
50         current_score = int(input("What is your score? "))
51     except ValueError:
52         # Error, can't turn what they typed into a number
53         print("I don't understand what you typed.")
54
55     # See if we have a new high score
56     if current_score > high_score:
57         # We do! Save to disk
58         print("Yea! New high score!")
59         save_high_score(current_score)
60     else:
61         print("Better luck next time.")
62
63 # Call the main function, start up the game
64 if __name__ == "__main__":
65     main()

```

1.28.4 Exception Objects

More information about an error can be pulled from the *exception object*. This object can be retrieved while catching an error using the `as` keyword. For example:

Listing 148: Creating an exception

```

1 try:
2     x = 5 / 0
3 except ZeroDivisionError as e:
4     print(e)

```

The `e` variable points to more information about the exception that can be printed out. More can be done with exceptions objects, but unfortunately that is beyond the scope of this chapter. Check the Python documentation online for more information about the exception object.

1.28.5 Exception Generating

Exceptions may be generated with the `raise` command. For example:

Listing 149: Creating an exception

```

1 # Generating exceptions
2 def get_input():
3     user_input = input("Enter something: ")
4     if len(user_input) == 0:
5         raise IOError("User entered nothing")

```

(continues on next page)

(continued from previous page)

```

6
7     get_input()

```

Try taking the code above, and add exception handling for the `IOError` raised.

It is also possible to create custom exceptions, but that is also beyond the scope of this book. Curious readers may learn more by going to:

<http://docs.python.org/tutorial/errors.html#raising-exceptions>

1.28.6 Proper Exception Use

Exceptions should not be used when `if` statements can just as easily handle the condition. Normal code should not raise exceptions when running the “happy path” scenario. Well-constructed try/catch code is easy to follow but code involving many exceptions and jumps in code to different handlers can be a nightmare to debug. (Once I was assigned the task of debugging code that read an XML document. It generated dozens of exceptions for each line of the file it read. It was incredibly slow and error-prone. That code should have never generated a single exception in the normal course of reading a file.)

1.29 Recursion

```

A child couldn't sleep, so her mother told her a story about a little frog,
  who couldn't sleep, so the frog's mother told her a story about a little bear,
    who couldn't sleep, so the bear's mother told her a story about a little weasel...
      who fell asleep.
        ...and the little bear fell asleep;
          ...and the little frog fell asleep;
            ...and the child fell asleep.

```

(Source: <http://everything2.com/title/recursion>)

Recursion is an object or process that is defined in terms of itself. Mathematical patterns such as factorials and the Fibonacci series are recursive. Documents that can contain other documents, which themselves can contain other documents, are recursive. Fractal images, and even certain biological processes are recursive in how they work.

1.29.1 Where is Recursion Used?

Documents, such as web pages, are naturally recursive. For example, Figure 20.1 shows a simple web document.

That web document can be contained in a “box,” which can help layout the page as shown in Figure 20.2.

This works recursively. Each box can contain a web page, that can have a box, which could contain another web page as shown in Figure 20.3.

Recursive functions are often used with advanced searching and sorting algorithms. We’ll show some of that here and if you take a “data structures” class you will see a lot more of it.

Even if a person does not become a programmer, understanding the concept of recursive systems is important. If there is a business need for recursive table structures, documents, or something else, it is important to know how to specify this to the programmer up front.

For example, a person might specify that a web program for recipes needs the ability to support ingredients and directions. A person familiar with recursion might state that each ingredient could itself be a recipes with other ingredients (that could be recipes.) The second system is considerably more powerful.

Web Page Title

Content for web page goes here Copyright © Info

Fig. 69: Figure 20.1: Web page



Fig. 70: Figure 20.2: Web page with tables



Fig. 71: Figure 20.3: Web page with recursion

1.29.2 How is Recursion Coded?

In prior chapters, we have used functions that call other functions. For example:

Listing 150: Functions calling other functions

```

1 def f():
2     g()
3     print("f")
4
5 def g():
6     print("g")
7
8 f()

```

It is also possible for a function to call itself. A function that calls itself is using a concept called recursion. For example:

Listing 151: Recursion

```

1 def f():
2     print("Hello")
3     f()
4
5 f()

```

The example above will print Hello and then call the `f()` function again. Which will cause another Hello to be printed out and another call to the `f()` function. This will continue until the computer runs out of something called stack space. When this happens, Python will output a long error that ends with:

RuntimeError: maximum recursion depth exceeded

The computer is telling you, the programmer, that you have gone too far down the rabbit hole.

1.29.3 Controlling Recursion Depth

To successfully use recursion, there needs to be a way to prevent the function from endlessly calling itself over and over again. The example below counts how many times it has been called, and uses an if statement to exit once the function has called itself ten times.

Listing 152: Controlling recursion levels

```
1 def f(level):
2     # Print the level we are at
3     print("Recursion call, level",level)
4     # If we haven't reached level ten...
5     if level < 10:
6         # Call this function again
7         # and add one to the level
8         f(level+1)
9
10 # Start the recursive calls at level 1
11 f(1)
```

Listing 153: Output

```
1 Recursion call, level 1
2 Recursion call, level 2
3 Recursion call, level 3
4 Recursion call, level 4
5 Recursion call, level 5
6 Recursion call, level 6
7 Recursion call, level 7
8 Recursion call, level 8
9 Recursion call, level 9
10 Recursion call, level 10
```

1.29.4 Recursion In Mathematics

Recursion Factorial Calculation

Any code that can be done recursively can be done without using recursion. Some programmers feel that the recursive code is easier to understand.

Calculating the factorial of a number is a classic example of using recursion. Factorials are useful in probability and statistics. For example:

Recursively, this can be described as:

Below are two example functions that calculate . The first one is non-recursive, the second is recursive.

Listing 154: Non-recursive factorial

```
1 # This program calculates a factorial
2 # WITHOUT using recursion
```

(continues on next page)

(continued from previous page)

```

3 def factorial_nonrecursive(n):
4     answer = 1
5     for i in range(2, n + 1):
6         answer = answer * i
7     return answer

```

Listing 155: Recursive factorial

```

1 # This program calculates a factorial
2 # WITH recursion
3 def factorial_recursive(n):
4     if n == 1:
5         return 1
6     elif n > 1:
7         return n * factorial_recursive(n - 1)

```

The functions do nothing by themselves. Below is an example where we put it all together. This example also adds some print statements inside the function so we can see what is happening.

Listing 156: Trying out recursive functions

```

1 # This program calculates a factorial
2 # WITHOUT using recursion
3
4 def factorial_nonrecursive(n):
5     answer = 1
6     for i in range(2, n + 1):
7         print(i, "*", answer, "=", i * answer)
8         answer = answer * i
9     return answer
10
11 print("I can calculate a factorial!")
12 user_input = input("Enter a number:")
13 n = int(user_input)
14 answer = factorial_nonrecursive(n)
15 print(answer)
16
17 # This program calculates a factorial
18 # WITH recursion
19
20 def factorial_recursive(n):
21     if n == 1:
22         return 1
23     else:
24         x = factorial_recursive(n - 1)
25         print(n, "*", x, "=", n * x)
26         return n * x
27
28 print("I can calculate a factorial!")
29 user_input = input("Enter a number:")
30 n = int(user_input)
31 answer = factorial_recursive(n)
32 print(answer)

```

Listing 157: Output

```

1 I can calculate a factorial!
2 Enter a number:7
3 2 * 1 = 2
4 3 * 2 = 6
5 4 * 6 = 24
6 5 * 24 = 120
7 6 * 120 = 720
8 7 * 720 = 5040
9 5040
10 I can calculate a factorial!
11 Enter a number:7
12 2 * 1 = 2
13 3 * 2 = 6
14 4 * 6 = 24
15 5 * 24 = 120
16 6 * 120 = 720
17 7 * 720 = 5040
18 5040

```

Recursive Expressions

Say you have a mathematical expression like this:

$$f_n = \begin{cases} 6 & \text{if } n = 1, \\ \frac{1}{2}f_{n-1} + 4 & \text{if } n > 1. \end{cases}$$

Looks complicated, but it just means that if $n = 1$ we are working with f_1 . That function returns a 6.

For f_2 we return $\frac{1}{2}f_1 + 4$.

The code would start with:

```

1 def f(n):

```

Then we need to add that first case:

```

1 def f(n):
2     if n == 1:
3         return 6

```

See how closely it follows the mathematical notation? Now for the rest:

```

1 def f(n):
2     if n == 1:
3         return 6
4     elif n > 1:
5         return (1 / 2) * f(n - 1) + 4

```

Converting these types of mathematical expressions to code is straight forward. But we'd better try it out in a full example:

```

1 def f(n):
2     if n == 1:
3         return 6

```

(continues on next page)

(continued from previous page)

```

4     elif n > 1:
5         return (1 / 2) * f(n - 1) + 4
6
7
8 def main():
9     result = f(10)
10    print(result)
11
12
13 main()

```

1.29.5 Recursive Graphics

Recursive Rectangles

Recursion is great to work with structured documents that are themselves recursive. For example, a web document can have a table divided into rows and columns to help with layout. One row might be the header, another row the main body, and finally the footer. Inside a table cell, might be another table. And inside of that can exist yet another table.

Another example is e-mail. It is possible to attach another person's e-mail to a your own e-mail. But that e-mail could have another e-mail attached to it, and so on.

Can we visually see recursion in action in one of our Pygame programs? Yes! Figure 19.4 shows an example program that draws a rectangle, and recursively keeps drawing rectangles inside of it. Each rectangle is 20% smaller than the parent rectangle. Look at the code. Pay close attention to the recursive call in the recursive_draw function.

Listing 158: recursive_rectangles.py

```

1 """
2 Recursive Rectangles
3 """
4 import arcade
5
6 SCREEN_WIDTH = 800
7 SCREEN_HEIGHT = 500
8
9
10 def draw_rectangle(x, y, width, height):
11     """ Recursively draw a rectangle, each one a percentage smaller """
12
13     # Draw it
14     arcade.draw_rectangle_outline(x, y, width, height, arcade.color.BLACK)
15
16     # As long as we have a width bigger than 1, recursively call this function with a_
17     # smaller rectangle
18     if width > 1:
19         # Draw the rectangle 90% of our current size
20         draw_rectangle(x, y, width * .9, height * .9)
21
22 class MyWindow(arcade.Window):
23     """ Main application class. """
24
25     def __init__(self, width, height):
26         super().__init__(width, height)

```

(continues on next page)

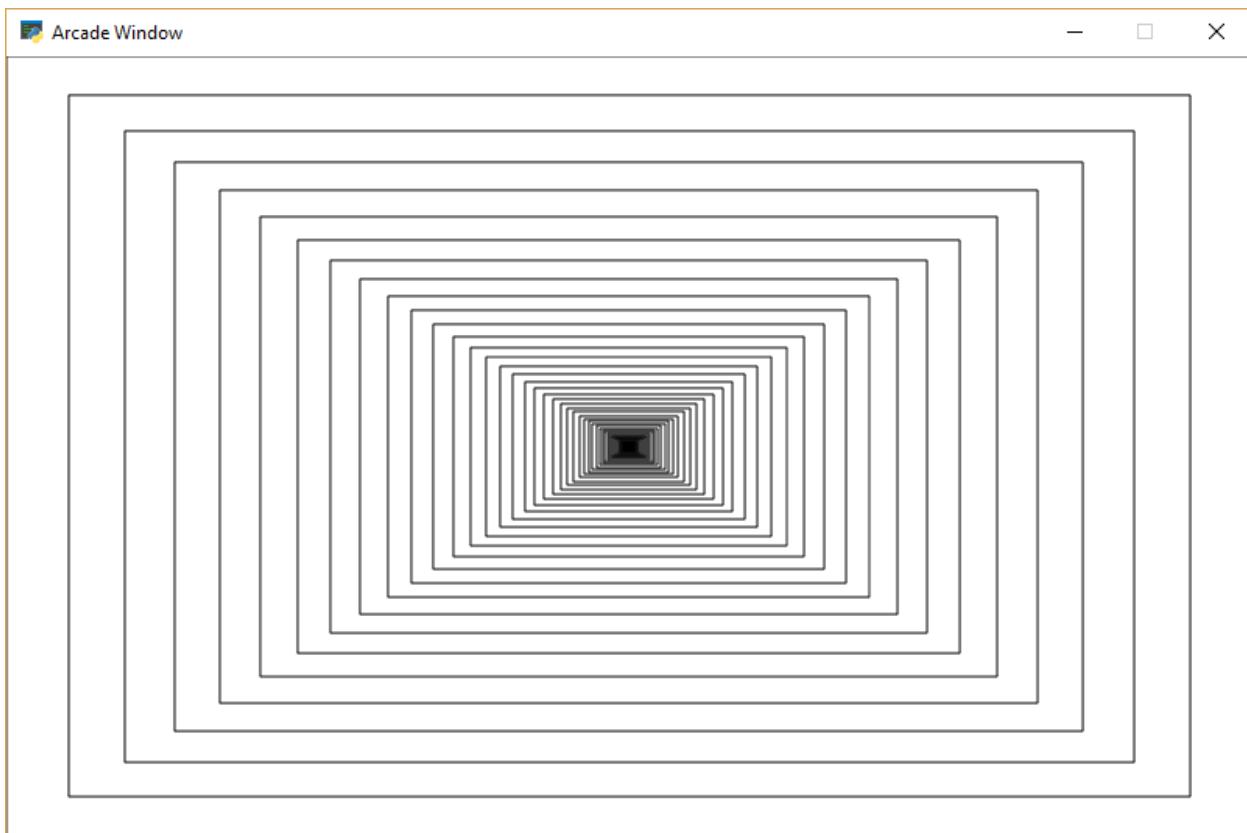


Fig. 72: Figure 20.4: Recursive Rectangles

(continued from previous page)

```

27     arcade.set_background_color(arcade.color.WHITE)
28
29
30     def on_draw(self):
31         """ Render the screen. """
32         arcade.start_render()
33
34         # Find the center of our screen
35         center_x = SCREEN_WIDTH / 2
36         center_y = SCREEN_HEIGHT / 2
37
38         # Start our recursive calls
39         draw_rectangle(center_x, center_y, SCREEN_WIDTH, SCREEN_HEIGHT)
40
41
42     def main():
43
44         MyWindow(SCREEN_WIDTH, SCREEN_HEIGHT)
45         arcade.run()
46
47
48 if __name__ == "__main__":
49     main()

```

Fractals

Fractals are defined recursively. Here is a very simple fractal, showing how it changes depending on how “deep” the recursion goes.

Here is the source code for the “H” fractal:

Listing 159: recursive_h.py

```

1 """
2 Recursive H's
3 """
4 import arcade
5
6 SCREEN_WIDTH = 800
7 SCREEN_HEIGHT = 500
8
9 RECURSION_DEPTH = 0
10
11
12 def draw_h(x, y, width, height, count):
13     """ Recursively draw an H, each one a half as big """
14
15     # Draw the H
16     # Draw cross-bar
17     arcade.draw_line(x + width * .25, height / 2 + y,
18                      x + width * .75, height / 2 + y, arcade.color.BLACK)
19     # Draw left side
20     arcade.draw_line(x + width * .25, height * .5 / 2 + y,
21                      x + width * .25, height * 1.5 / 2 + y, arcade.color.BLACK)
22     # Draw right side

```

(continues on next page)

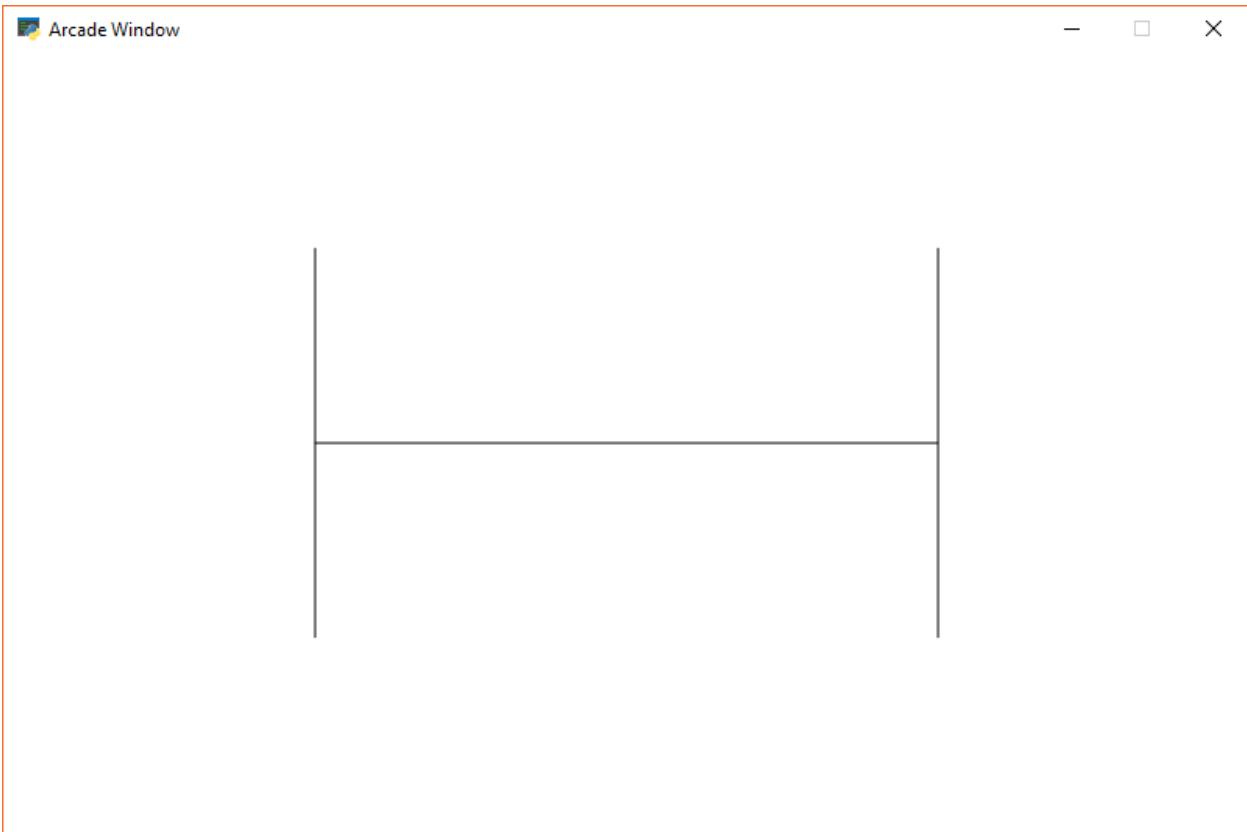


Fig. 73: Figure 20.5: Recursive Fractal Level 0

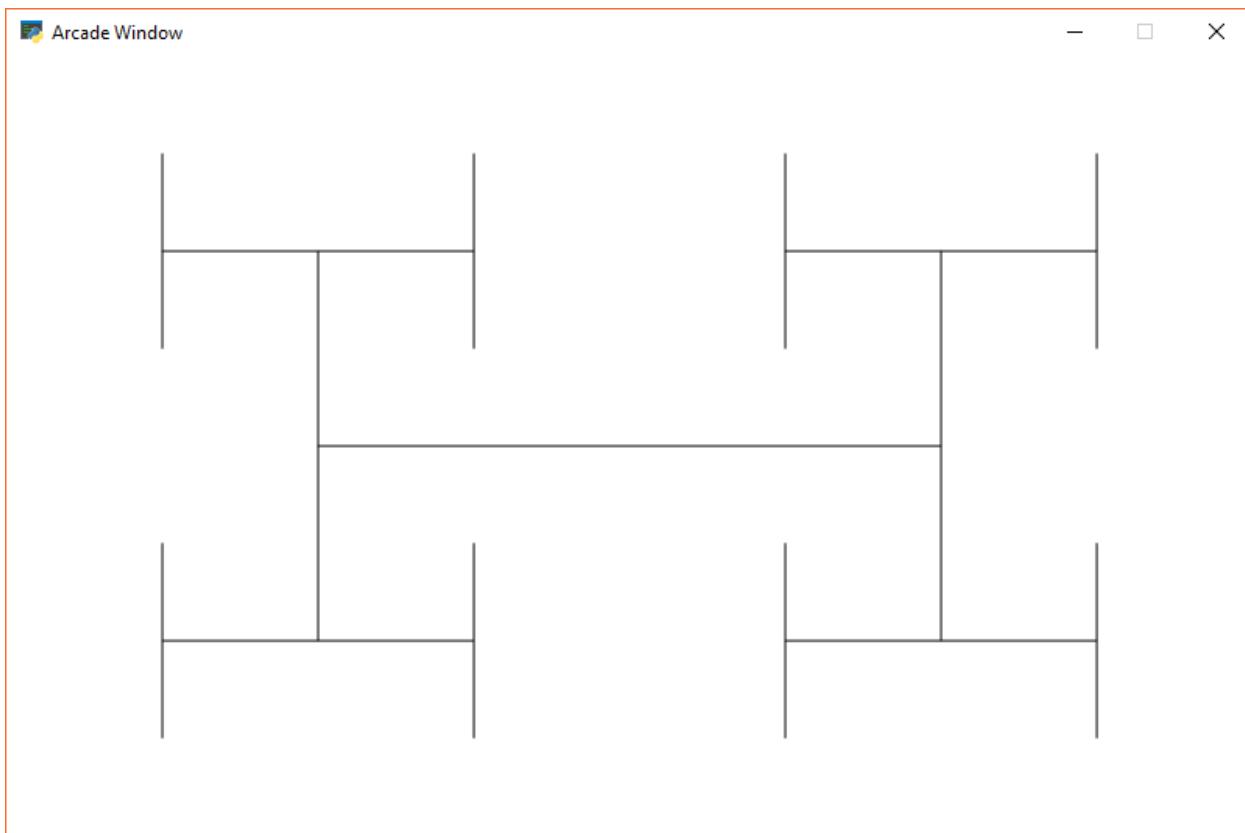


Fig. 74: Figure 20.6: Recursive Fractal Level 1

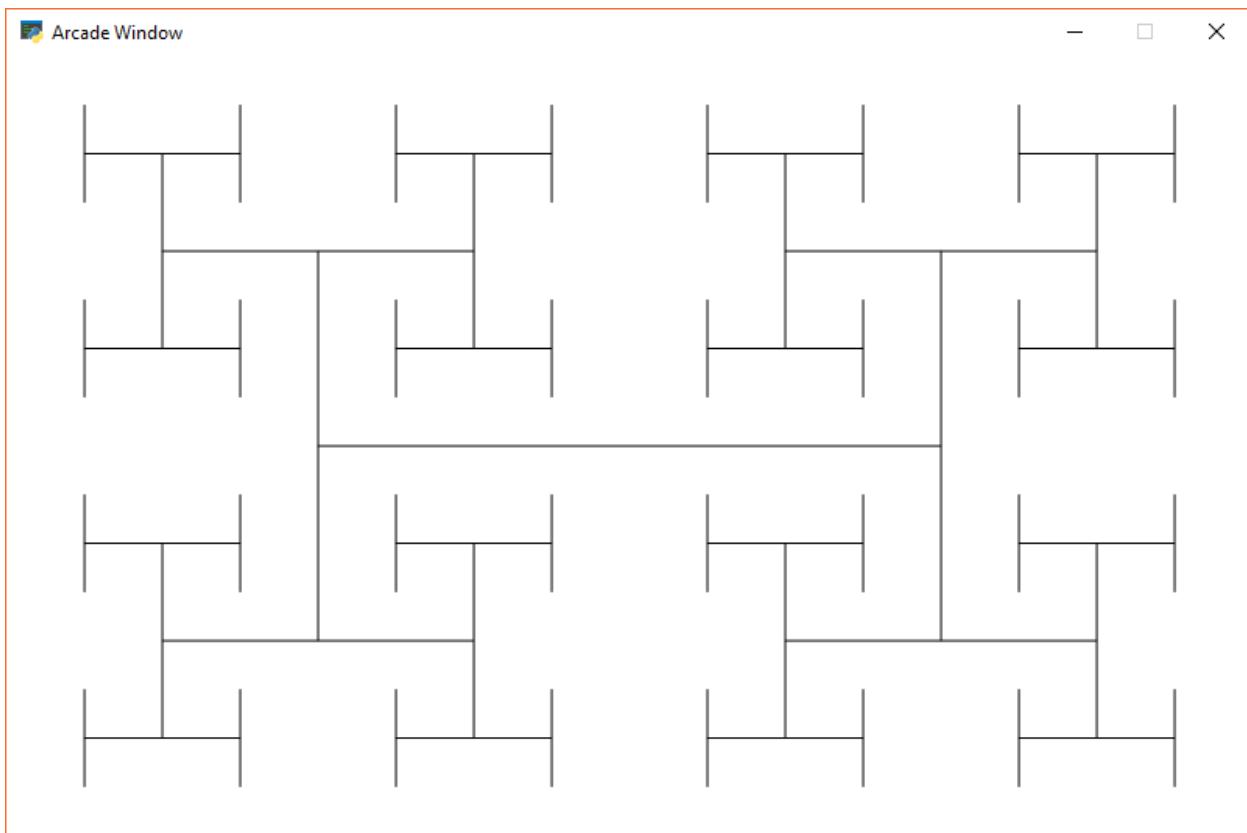


Fig. 75: Figure 20.7: Recursive Fractal Level 2

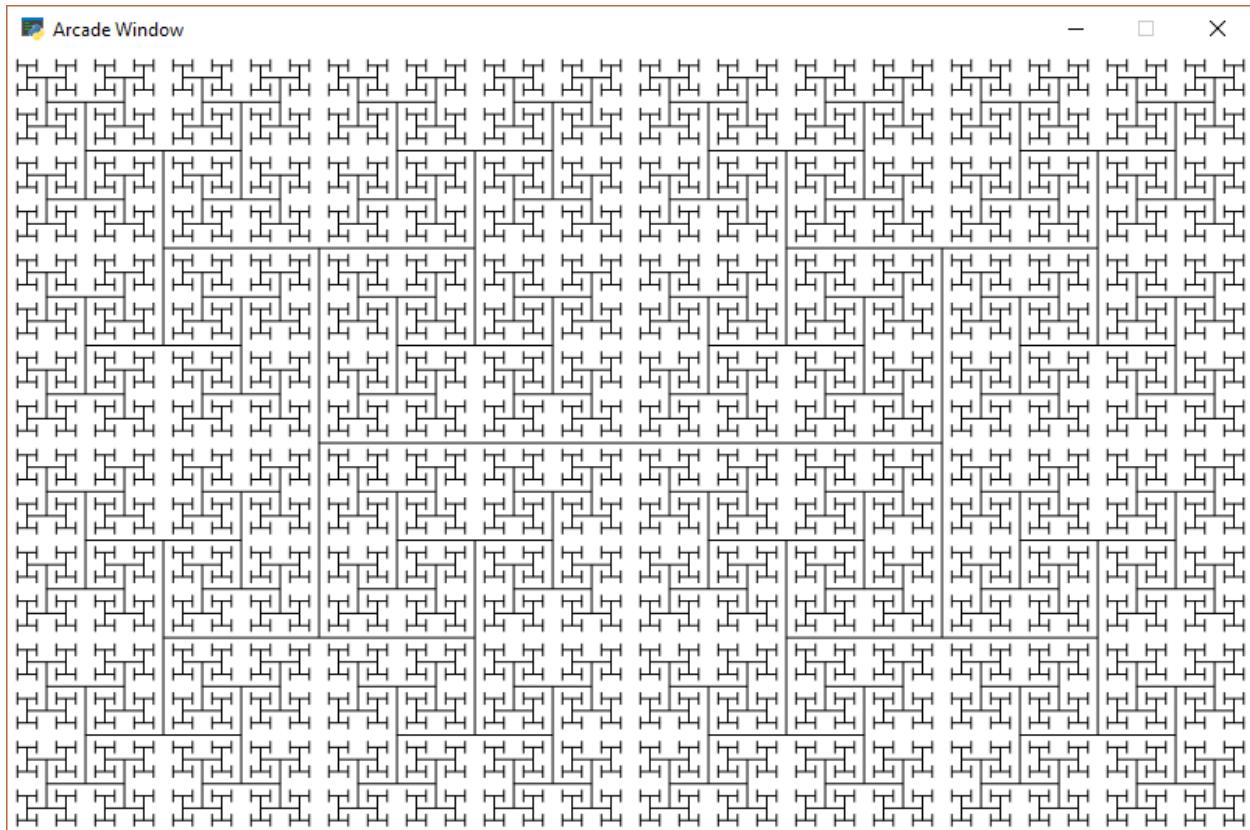


Fig. 76: Figure 20.8: Recursive Fractal Level 5

(continued from previous page)

```

23     arcade.draw_line(x + width * .75, height * .5 / 2 + y,
24                         x + width * .75, height * 1.5 / 2 + y, arcade.color.BLACK)
25
26     # As long as we have a width bigger than 1, recursively call this function with a
27     # smaller rectangle
28     if count > 0:
29         count -= 1
30         # Draw the rectangle 90% of our current size
31         # Draw lower left
32         draw_h(x, y, width / 2, height / 2, count)
33         # Draw lower right
34         draw_h(x + width / 2, y, width / 2, height / 2, count)
35         # Draw upper left
36         draw_h(x, y + height / 2, width / 2, height / 2, count)
37         # Draw upper right
38         draw_h(x + width / 2, y + height / 2, width / 2, height / 2, count)
39
40 class MyWindow(arcade.Window):
41     """ Main application class. """
42
43     def __init__(self, width, height):
44         super().__init__(width, height)
45
46         arcade.set_background_color(arcade.color.WHITE)
47
48     def on_draw(self):
49         """ Render the screen. """
50         arcade.start_render()
51
52         # Start our recursive calls
53         draw_h(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, RECURSION_DEPTH)
54
55
56 def main():
57     MyWindow(SCREEN_WIDTH, SCREEN_HEIGHT)
58     arcade.run()
59
60
61 if __name__ == "__main__":
62     main()

```

You can explore fractals on-line:

- <https://www.chromeexperiments.com/fractal>
- <http://usefuljs.net/fractals/>
- <http://hirnsohle.de/test/fractalLab/>

If you want to program your own fractals, you can get ideas of easy fractals by looking at Chapter 8 of [The Nature of Code](#) by Daniel Shiffman.

1.29.6 Recursive Mazes

There are maze generation algorithms. Wikipedia has a nice [Maze](#) generation algorithm article that details some. One way is the *recursive division method*.

The algorithm is described below. Images are from Wikipedia.

Fig. 77: Begin with the maze's space with just the outside walls. Call this a chamber.

Fig. 78: Divide the chamber with one random horizontal wall, and one random vertical wall.

Fig. 79: Pick three of the four walls and put a randomly position opening in it.

This method results in mazes with long straight walls crossing their space, making it easier to see which areas to avoid.

Here is sample Python code that creates a maze using this method:

Listing 160: Recursive Maze Example

```

1 import random
2
3 # These constants are used to determine what should be stored in the grid if we have ↵
4 # an empty
5 # space or a filled space.
6 EMPTY = " "
7 WALL = "XXX"
8
9 # Maze must have an ODD number of rows and columns.
10 # Walls go on EVEN rows/columns.
11 # Openings go on ODD rows/columns
12 MAZE_HEIGHT = 51
13 MAZE_WIDTH = 51
14
15 def create_grid(width, height):
16     """ Create an empty grid. """
17     grid = []
18     for row in range(height):
19         grid.append([])
20         for column in range(width):
21             grid[row].append(EMPTY)
22     return grid
23
24
25 def print_maze(maze):
26     """ Print the maze. """
27
28     # Loop each row, but do it in reverse so 0 is at the bottom like we expect
29     for row in range(len(maze) - 1, -1, -1):
30         # Print the row/y number
31         print(f"{row:3} - ", end="")
32
33         # Loop the row and print the content
34         for column in range(len(maze[row])):
35             print(f"{maze[row][column]}", end="")
36
37         # Go down a line
38         print()
39

```

(continues on next page)

Fig. 80: Subdivide each of the four sections. Call each a chamber. Recursively call this function. In this image, the top left chamber has been subdivided.

Fig. 81: Finished maze.

(continued from previous page)

```

40     # Print the column/x at the bottom
41     print("      ", end="")
42     for column in range(len(maze[0])):
43         print(f"{column:3}", end="")
44     print()
45
46
47 def create_outside_walls(maze):
48     """ Create outside border walls."""
49
50     # Create left and right walls
51     for row in range(len(maze)):
52         maze[row][0] = WALL
53         maze[row][len(maze[row])-1] = WALL
54
55     # Create top and bottom walls
56     for column in range(1, len(maze[0]) - 1):
57         maze[0][column] = WALL
58         maze[len(maze[0]) - 1][column] = WALL
59
60
61 def create_maze(maze, top, bottom, left, right):
62     """
63         Recursive function to divide up the maze in four sections
64         and create three gaps.
65         Walls can only go on even numbered rows/columns.
66         Gaps can only go on odd numbered rows/columns.
67         Maze must have an ODD number of rows and columns.
68     """
69
70     # Figure out where to divide horizontally
71     start_range = bottom + 2
72     end_range = top - 1
73     y = random.randrange(start_range, end_range, 2)
74
75     # Do the division
76     for column in range(left + 1, right):
77         maze[y][column] = WALL
78
79     # Figure out where to divide vertically
80     start_range = left + 2
81     end_range = right - 1
82     x = random.randrange(start_range, end_range, 2)
83

```

(continues on next page)

Fig. 82: Recursive maze generation. Source: Wikipedia [Maze generation algorithm](#).

(continued from previous page)

```

84     # Do the division
85     for row in range(bottom + 1, top):
86         maze[row][x] = WALL
87
88     # Now we'll make a gap on 3 of the 4 walls.
89     # Figure out which wall does NOT get a gap.
90     wall = random.randrange(4)
91     if wall != 0:
92         gap = random.randrange(left + 1, x, 2)
93         maze[y][gap] = EMPTY
94
95     if wall != 1:
96         gap = random.randrange(x + 1, right, 2)
97         maze[y][gap] = EMPTY
98
99     if wall != 2:
100        gap = random.randrange(bottom + 1, y, 2)
101        maze[gap][x] = EMPTY
102
103    if wall != 3:
104        gap = random.randrange(y + 1, top, 2)
105        maze[gap][x] = EMPTY
106
107    # Print what's going on
108    print(f"Top/Bottom: {top}, {bottom} Left/Right: {left}, {right} Divide: {x}, {y}")
109    print_maze(maze)
110    print()
111
112    # If there's enough space, to a recursive call.
113    if top > y + 3 and x > left + 3:
114        create_maze(maze, top, y, left, x)
115
116    if top > y + 3 and x + 3 < right:
117        create_maze(maze, top, y, x, right)
118
119    if bottom + 3 < y and x + 3 < right:
120        create_maze(maze, y, bottom, x, right)
121
122    if bottom + 3 < y and x > left + 3:
123        create_maze(maze, y, bottom, left, x)
124
125
126 def main():
127
128     # Create the blank grid
129     maze = create_grid(MAZE_WIDTH, MAZE_HEIGHT)
130
131     # Fill in the outside walls
132     create_outside_walls(maze)
133
134     # Start the recursive process
135     create_maze(maze, MAZE_HEIGHT - 1, 0, 0, MAZE_WIDTH - 1)
136
137
138 if __name__ == "__main__":
139     main()

```

1.29.7 Recursive Binary Search

Recursion can be also be used to perform a binary search. Here is a non-recursive binary search from Chapter 15:

Listing 161: Non-recursive binary search

```
1 def binary_search_nonrecursive(search_list, key):
2     lower_bound = 0
3     upper_bound = len(search_list) - 1
4     found = False
5     while lower_bound < upper_bound and found == False:
6         middle_pos = (lower_bound + upper_bound) // 2
7         if search_list[middle_pos] < key:
8             lower_bound = middle_pos + 1
9         elif search_list[middle_pos] > key:
10            upper_bound = middle_pos
11        else:
12            found = True
13
14    if found:
15        print("The name is at position", middle_pos)
16    else:
17        print("The name was not in the list." )
18
19 binary_search_nonrecursive(name_list, "Morgiana the Shrew")
```

This same binary search written in a recursive manner:

Listing 162: Recursive binary search

```
1 def binary_search_recursive(search_list, key, lower_bound, upper_bound):
2     middle_pos = (lower_bound + upper_bound) // 2
3     if search_list[middle_pos] < key:
4         # Recursively search top half
5         binary_search_recursive(search_list, key,
6                                middle_pos + 1, upper_bound)
7     elif search_list[middle_pos] > key:
8         # Recursively search bottom half
9         binary_search_recursive(search_list, key,
10                                lower_bound, middle_pos )
11    else:
12        print("Found at position", middle_pos)
13
14 lower_bound = 0
15 upper_bound = len(name_list) - 1
16 binary_search_recursive(name_list, "Morgiana the Shrew",
17                         lower_bound, upper_bound)
```

1.30 String Formatting

Looking for an easy way to format variables for display, and mix them with other text? Add thousands separators (commas in the US) to a large number, and format decimals the way you want them? You need “string formatting.”

Python has three ways to format strings. The old string formatting which originally came from how older languages such as C did formatting. The newer way introduced in Python 3 that was supposed to be better, but arguably wasn’t. And the new way introduced in Python 3.6 with [PEP-498](#). We’ll cover the newest way.

1.30.1 Quick Reference

Here is a quick table for reference when doing text formatting. For a detailed explanation of how text formatting works, keep reading.

Table 1: Example Formatting Commands

Number	Format	Output	Description
x = 3.1415926	print(f"{x:3.14 2f}")		2 decimal places
x = 3.1415926	print(f"{x:+3.14 2f}")		2 decimal places with sign
x = -1	print(f"{x:+1.00 2f}")		2 decimal places with sign
x = 3.1415926	print(f"{x:3 0f}")		No decimal places (will round)
x = 5	print(f"{x:@5>2d}")		Pad with zeros on the left
x = 1000000	print(f"{x:1,000,000 }")		Number format with comma separator
x = 0.25	print(f"{x:25.00% 2%}")		Format percentage
x = 1000000000	print(f"{x:1.00e+09 2e}")		Exponent notation
x = 11	print(f">{x:>10d}").11		Right aligned
x = 11	print(f">{x:<10d}")....		Left aligned
x = 11	print(f">{x:^10d}")....		Center aligned

1.30.2 Decimal Numbers

Try running the following program, which prints out several random numbers.

Listing 163: Print an unformatted list of numbers

```

1 import random
2
3 for i in range(10):
4     x = random.randrange(120)
5     print("My number: ", x)

```

The output is left justified and numbers look terrible:

```

My number: 30
My number: 2
My number: 101
My number: 3
My number: 44
My number: 111
My number: 100
My number: 48
My number: 27
My number: 92

```

We can use string formatting to make the list of numbers look better by right-justifying them. The first step is to use the “Literal String Interpolation” on the string. See below:

```
import random

for i in range(10):
    x = random.randrange(120)
    print(f"My number: {x}")
```

This gets our program closer to right-justifying the number, but we aren't quite there yet. See how the string starts with `f`?

The string will not print out the curly braces `{}` but instead replace them with the value in `x`. The output (below) looks just like what we had before.

Listing 164: The output:

```
My number: 23
My number: 92
My number: 102
My number: 19
My number: 85
My number: 114
My number: 37
My number: 101
My number: 35
My number: 18
```

To right justify, we add more information about how to format the number to the curly braces `{}`:

Listing 165: Right justified list of numbers

```
1 import random
2
3 for i in range(10):
4     x = random.randrange(120)
5     print(f"My number: {x:3}")
```

Listing 166: The output:

```
My number: 37
My number: 108
My number: 117
My number: 55
My number: 19
My number: 97
My number: 78
My number: 12
My number: 29
My number: 0
```

This is better; we have right justified numbers! But how does it work? The `:3` that we added isn't exactly intuitive. Looks like we just added a random emoji.

Here's the breakdown: The `{ }` tells the computer we are going to format a number. Inside we put the variable we want to format, `x` in this case. After the variable, we put a `:` to tell the computer we are about to give it formatting information.

In this case we give it a `3` to specify a field width of three characters. The field width value tells the computer to try to fit the number into a field three characters wide. By default, it will try to right-justify numbers and left-justify text.

Even better, the program no longer needs to call `str()` to convert the number to a string! Leave the string conversions out.

What if you had large numbers? Let's make bigger random numbers:

Listing 167: Bigger numbers that are hard to read

```
1 import random
2
3 for i in range(10):
4     x = random.randrange(100000)
5     print(f"My number: {x:6}")
```

This gives output that is right justified, but still doesn't look good.

Listing 168: The output:

```
My number: 89807
My number: 5177
My number: 24067
My number: 19887
My number: 54155
My number: 49288
My number: 31412
My number: 49633
My number: 43406
My number: 37398
```

Where are the commas? This list would look better with separators between each three digits. Take a look at the next example to see how they are added in:

Listing 169: Adding a thousands separator

```
1 import random
2
3 for i in range(10):
4     x = random.randrange(100000)
5     print(f"My number: {x:6,}")
```

The output:

```
My number: 86,631
My number: 57,165
My number: 19,835
My number: 22,560
My number: 43,161
My number: 16,604
My number: 20,544
My number: 33,906
My number: 89,846
My number: 27,350
```

We added a comma after the field width specifier, and now our numbers have commas. That comma must go after the field width specifier, not before. Commas are included in calculating the field width. For example, 1,024 has a field width of 5, not 4.

We can print multiple values, and combine the values with text. Run the code below.

Listing 170: Printing more than one variable at a time

```
1 x = 5
2 y = 66
3 z = 777
4 print(f"A = '{x}' B = '{y}' C = '{z}'")
```

The program will substitute numbers in for the curly braces, and still print out all of the other text in the string:

```
A = '5' B = '66' C = '777'
```

1.30.3 Strings

Let's look at how to format strings.

The following list looks terrible.

Listing 171: Terrible looking list

```
1 my_fruit = ["Apples", "Oranges", "Grapes", "Pears"]
2 my_calories = [4, 300, 70, 30]
3
4 for i in range(4):
5     print(my_fruit[i], "are", my_calories[i], "calories.")
```

The output:

```
Apples are 4 calories.
Oranges are 300 calories.
Grapes are 70 calories.
Pears are 30 calories.
```

Now try it using the format command. Note how we can put additional text and more than one value into the same line.

Listing 172: Formatting a list of fruit

```
1 my_fruit = ["Apples", "Oranges", "Grapes", "Pears"]
2 my_calories = [4, 300, 70, 30]
3
4 for i in range(4):
5     print(f"{my_fruit[i]}: {my_calories[i]} calories.")
```

Listing 173: The output:

```
Apples are 4 calories.
Oranges are 300 calories.
Grapes are 70 calories.
Pears are 30 calories.
```

That's pretty cool, and it looks the way we want it. But what if we didn't want the numbers right justified, and the text left justified? We can use the < and > characters like the following example:

Listing 174: Specifying right/left alignment

```

1 my_fruit = ["Apples", "Oranges", "Grapes", "Pears"]
2 my_calories = [4, 300, 70, 30]
3
4 for i in range(4):
5     print(f"{my_fruit[i]:>7} are {my_calories[i]:<3} calories.")

```

Listing 175: The output:

```

Apples are 4    calories.
Oranges are 300 calories.
Grapes are 70   calories.
Pears are 30    calories.

```

1.30.4 Leading Zeros

This produces output that isn't right:

Listing 176: Terrible looking clock

```

1 for hours in range(1,13):
2     for minutes in range(0,60):
3         print(f"Time {hours}:{minutes}")

```

Listing 177: The not-very-good output:

```

Time 8:56
Time 8:57
Time 8:58
Time 8:59
Time 9:0
Time 9:1
Time 9:2

```

We need to use leading zeros for displaying numbers in clocks. Rather than specify a 2 for the field width, instead use 02. This will pad the field with zeros rather than spaces.

Listing 178: Formatting time output with leading zeros

```

1 for hours in range(1, 13):
2     for minutes in range(0, 60):
3         print(f"Time {hours:02}:{minutes:02}")

```

Listing 179: The output:

```

Time 08:56
Time 08:57
Time 08:58
Time 08:59
Time 09:00
Time 09:01
Time 09:02

```

1.30.5 Floating Point Numbers

We can also control floating point output. Examine the following code and its output:

Listing 180: Formatting float point numbers

```

1 x = 0.1
2 y = 123.456789
3
4 print(f"{x:.1} {y:.1}")
5 print(f"{x:.2} {y:.2}")
6 print(f"{x:.3} {y:.3}")
7 print(f"{x:.4} {y:.4}")
8 print(f"{x:.5} {y:.5}")
9 print(f"{x:.6} {y:.6}")
10
11 print()
12 print(f"{x:.1f} {y:.1f}")
13 print(f"{x:.2f} {y:.2f}")
14 print(f"{x:.3f} {y:.3f}")
15 print(f"{x:.4f} {y:.4f}")
16 print(f"{x:.5f} {y:.5f}")
17 print(f"{x:.6f} {y:.6f}")

```

Listing 181: And here's the output for that code:

```

1 0.1 1e+02
2 0.1 1.2e+02
3 0.1 1.23e+02
4 0.1 123.5
5 0.1 123.46
6 0.1 123.457
7
8 0.1 123.5
9 0.10 123.46
10 0.100 123.457
11 0.1000 123.4568
12 0.10000 123.45679
13 0.100000 123.456789

```

A format of `.2` means to display the number with two digits of precision. Unfortunately this means if we display the number `123` which has three significant numbers rather than rounding it we get the number in scientific notation: `1.2e+02`.

A format of `.2f` (note the `f`) means to display the number with two digits after the decimal point. So the number `1` would display as `1.00` and the number `1.5555` would display as `1.56`.

A program can also specify a field width character:

Listing 182: Specifying a field width character

```

1 x = 0.1
2 y = 123.456789
3
4 print(f"My number: '{x:10.1}' and '{y:10.1}'")
5 print(f"My number: '{x:10.2}' and '{y:10.2}'")
6 print(f"My number: '{x:10.3}' and '{y:10.3}'")
7 print(f"My number: '{x:10.4}' and '{y:10.4}'")

```

(continues on next page)

(continued from previous page)

```

8 print(f"My number: '{x:10.5}' and '{y:10.5}'")
9 print(f"My number: '{x:10.6}' and '{y:10.6}'")
10
11 print()
12 print(f"My number: '{x:10.1f}' and '{y:10.1f}'")
13 print(f"My number: '{x:10.2f}' and '{y:10.2f}'")
14 print(f"My number: '{x:10.3f}' and '{y:10.3f}'")
15 print(f"My number: '{x:10.4f}' and '{y:10.4f}'")
16 print(f"My number: '{x:10.5f}' and '{y:10.5f}'")
17 print(f"My number: '{x:10.6f}' and '{y:10.6f}'")

```

The format `10.2f` does not mean 10 digits before the decimal and two after. It means a total field width of 10. So there will be 7 digits before the decimal, the decimal which counts as one more, and 2 digits after.

Listing 183: The output:

```

My number: '          0.1' and '      1e+02'
My number: '          0.1' and '      1.2e+02'
My number: '          0.1' and '      1.23e+02'
My number: '          0.1' and '      123.5'
My number: '          0.1' and '      123.46'
My number: '          0.1' and '      123.457'

My number: '          0.1' and '      123.5'
My number: '          0.10' and '     123.46'
My number: '          0.100' and '    123.457'
My number: '          0.1000' and '   123.4568'
My number: '          0.10000' and ' 123.45679'
My number: '          0.100000' and '123.456789'

```

1.30.6 Printing Dollars and Cents

If you want to print a floating point number for cost, you use an f. See below:

Listing 184: Specifying a field width character

```

1 cost1 = 3.07
2 tax1 = cost1 * 0.06
3 total1 = cost1 + tax1
4
5 print(f"Cost: ${cost1:5.2f}")
6 print(f"Tax:      {tax1:5.2f}")
7 print(f"-----")
8 print(f"Total: ${total1:5.2f}")

```

Remember! It would be easy to think that `%5.2f` would mean five digits, a decimal, followed by two digits. But it does not. It means a total field width of five, including the decimal and the two digits after. Here's the output:

Listing 185: The output:

```
Cost: $ 3.07
Tax:    0.18
-----
Total: $ 3.25
```

Danger! The above code has a mistake that is very common when working with financial transactions. Can you spot it? Try spotting it with the expanded code example below:

Listing 186: Specifying a field width character

```
1 cost1 = 3.07
2 tax1 = cost1 * 0.06
3 total1 = cost1 + tax1
4
5 print(f"Cost: ${cost1:5.2f}")
6 print(f"Tax:     {tax1:5.2f}")
7 print(f"-----")
8 print(f"Total: ${total1:5.2f}")
9
10 cost2 = 5.07
11 tax2 = cost2 * 0.06
12 total2 = cost2 + tax2
13
14 print()
15 print(f"Cost: ${cost2:5.2f}")
16 print(f"Tax:     {tax2:5.2f}")
17 print(f"-----")
18 print(f"Total: ${total2:5.2f}")
19
20 print()
21 grand_total = total1 + total2
22 print(f"Grand total: ${grand_total:5.2f}")
```

Listing 187: The output:

```
Cost: $ 3.07
Tax:    0.18
-----
Total: $ 3.25

Cost: $ 5.07
Tax:    0.30
-----
Total: $ 5.37

Grand total: $ 8.63
```

Spot the mistake? You have to watch out for rounding errors! Look at that example, it seems like the total should be \$ 8.62 but it isn't.

Print formatting doesn't change the number, only what is output! If we changed the print formatting to include three digits after the decimal the reason for the error becomes more apparent:

Listing 188: The output:

```
Cost: $3.070
Tax: 0.184
-----
Total: $3.254

Cost: $5.070
Tax: 0.304
-----
Total: $5.374

Grand total: $8.628
```

Again, formatting for the display does not change the number. Use the round command to change the value and truly round. See below:

Listing 189: Specifying a field width character

```
1 cost1 = 3.07
2 tax1 = round(cost1 * 0.06, 2)
3 total1 = cost1 + tax1
4
5 print(f"Cost: ${cost1:5.2f}")
6 print(f"Tax: {tax1:5.2f}")
7 print(f"-----")
8 print(f"Total: ${total1:5.2f}")
9
10 cost2 = 5.07
11 tax2 = round(cost2 * 0.06, 2)
12 total2 = cost2 + tax2
13
14 print()
15 print(f"Cost: ${cost2:5.2f}")
16 print(f"Tax: {tax2:5.2f}")
17 print(f"-----")
18 print(f"Total: ${total2:5.2f}")
19
20 print()
21 grand_total = total1 + total2
22 print(f"Grand total: ${grand_total:5.2f}")
```

Listing 190: The output:

```
Cost: $ 3.07
Tax: 0.18
-----
Total: $ 3.25

Cost: $ 5.07
Tax: 0.30
-----
Total: $ 5.37

Grand total: $ 8.62
```

The round command controls how many digits after the decimal we round to. It returns the rounded value but does not

change the original value. See below:

Listing 191: Specifying a field width character

```
1 x = 1234.5678
2 print(round(x, 2))
3 print(round(x, 1))
4 print(round(x, 0))
5 print(round(x, -1))
6 print(round(x, -2))
```

See below to figure out how feeding the round() function values like -2 for the digits after the decimal affects the output:

Listing 192: The output:

```
1234.57
1234.6
1235.0
1230.0
1200.0
```

1.30.7 Use in Arcade Programs

We don't just have to format strings for print statements. The example `timer.py` uses string formatting to make an on-screen timer:

Listing 193: Code from `timer.py`

```
1 def on_draw(self):
2     """ Use this function to draw everything to the screen. """
3
4     # Start the render. This must happen before any drawing
5     # commands. We do NOT need an stop render command.
6     arcade.start_render()
7
8     # Calculate minutes
9     minutes = int(self.total_time) // 60
10
11    # Calculate seconds by using a modulus (remainder)
12    seconds = int(self.total_time) % 60
13
14    # Figure out our output
15    output = f"Time: {minutes:02d}:{seconds:02d}"
16
17    # Output the timer text.
18    arcade.draw_text(output, 300, 300, arcade.color.BLACK, 30)
19
20 def update(self, delta_time):
21     """
22     All the logic to move, and the game logic goes here.
23     """
24     self.total_time += delta_time
```

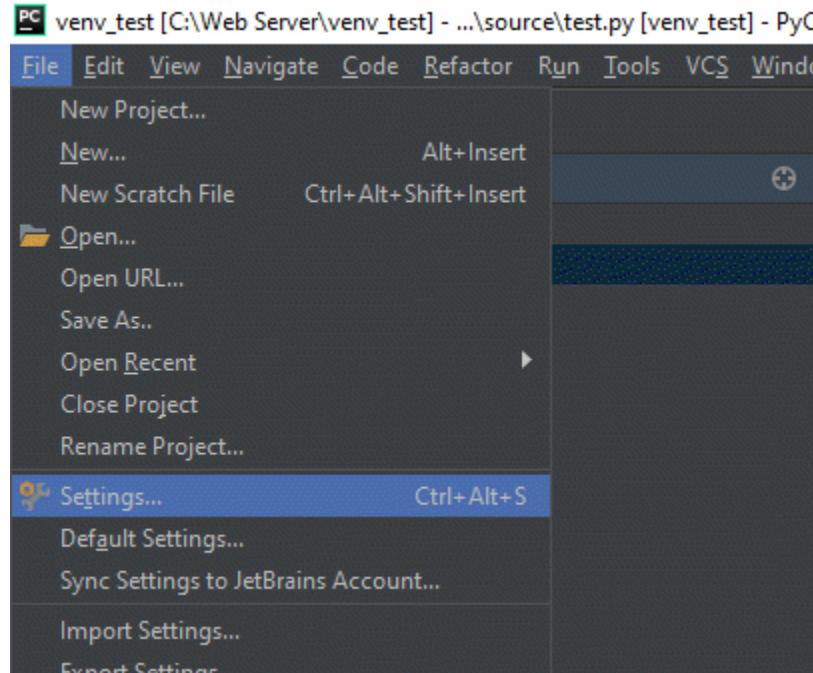
You can also use it to display the score, or any other statistics you'd like to show the player.

1.31 Setting Up a Virtual Environment In PyCharm

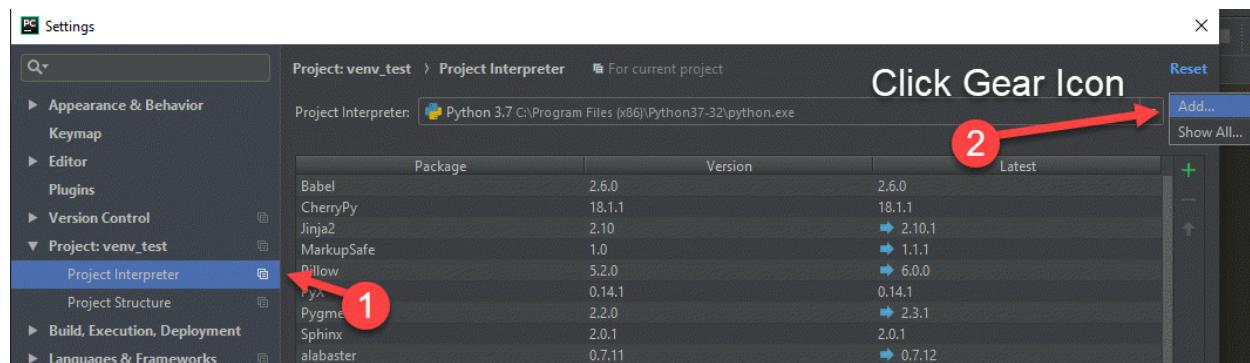
A Python virtual environment (venv) allows libraries to be installed for just a single project, rather than shared across everyone using the computer. It also does not require administrator privileges to install.

Assuming you already have a project, follow these steps to create a venv:

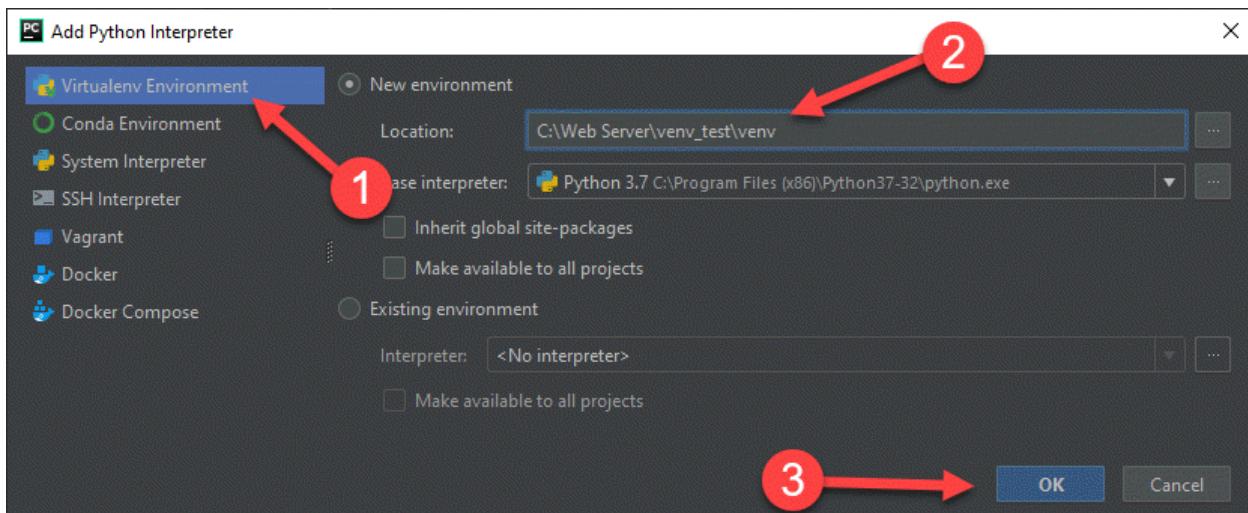
Step 1: Select File... Settings



Step 2: Click “Python Interpreter”. Then find the gear icon in the upper right. click on it and select “Add”

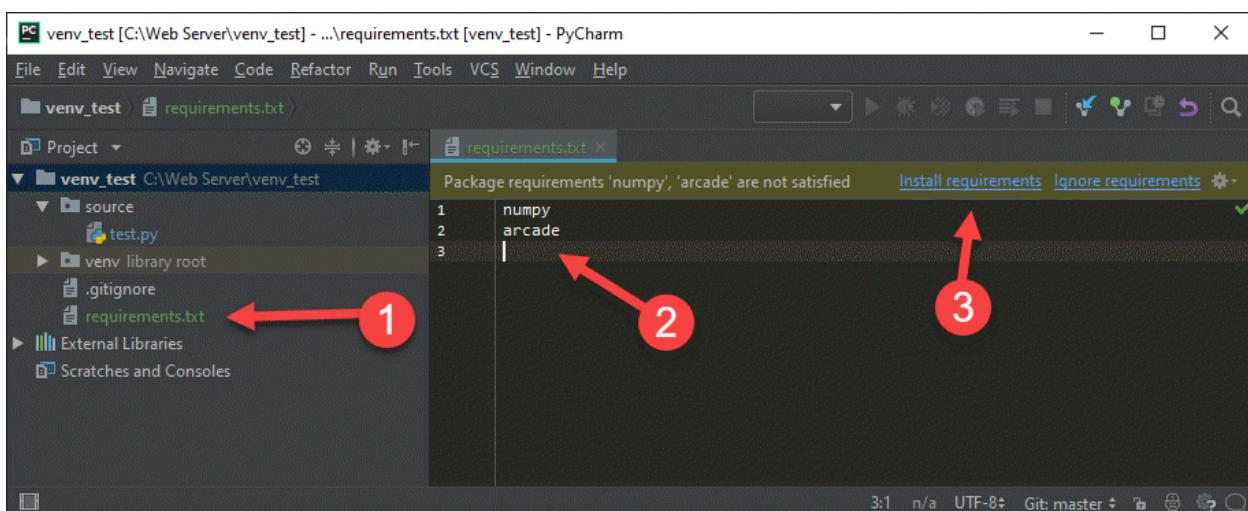


Step 3: Select Virtualenv Environment from the left. Then create a new environment. Usually it should be in a folder called venv in your main project. PyCharm does not always select the correct location by default, so carefully look at the path to make sure it is correct, then select “Ok”.



Now a virtual environment has been set up. The standard in Python projects is to create a file called `requirements.txt` and list the packages you want in there.

PyCharm will automatically ask if you want to install those packages as soon as you type them in. Go ahead and let it.



CHAPTER 2

Labs

2.1 Lab 1: First Program

Follow these step-by-step instructions. Note that a lot of the instructions link to a different site. I'd recommend opening those links in a different tab on your browser. That way you don't lose your spot on what lab step you are on.

1. If you are installing on your own computer:
 1. Install Python and set up the libraries. See [Installing Python and Arcade](#).
 2. Install PyCharm. See [Installing an IDE](#).
 3. Install Git. See [Installing Git](#).
4. Fork the repository. See [Forking the Repository](#).
5. Share the repository. See [Share the Repository](#).
6. Clone the repository. See [Cloning the Repository](#).
7. Open your project in PyCharm. See [Open Project in Pycharm](#).
8. Try changing a file. See [Change a File](#).
9. Commit and push the updated program to BitBucket. See [Commit Your Code](#).
10. Great! Now let's try programming something. Create and run a test program that prints Hello. See [Printing Hello World](#). Make sure you take the time to learn how to "run" the program and see the output.
11. Update your program to print something longer. Commit and push again. See [Multiple Print Lines](#).
12. Examine and try out different **escape codes**. See [Escape Codes](#). (We'll talk more about this Friday.)
13. Commit and push the updated program to BitBucket. See [Commit Your Code](#) again.
14. Now is the time to work on Lab 01. Make sure you are working in the Lab 01 folder, with the `lab_01.py` file.
15. Make the lab *yours*. Figure out something of your own to print. Don't just use my examples. Make it longer than one line. At least four. Also include at least one escape code.

16. Try running your lab. If you see the old “test” program running instead, right click on your program and tell it you want it to run `lab_01.py` not the other program.
17. Commit and push your lab.
18. Find your lab in BitBucket. Go to “source” and find the directory it is in. Copy the URL, submit for grading.

Before turning in your program, double-check:

- Did you put the code in `lab_01.py` in the Lab 1 folder?
- Did you remove the sample prints shown above, and make up your own print statements? At least four lines?
- Did you invite your instructor to the repository?
- Does the program use proper spelling, capitalization, and grammar in the text that you printed to the screen? (Seriously, make sure what you print out has periods and capital letters.)
- Does the program use at least one escape code? Also, remember you don’t have to put spaces around escape codes. So do this: "`\tIndented with a tab`" not "`\t Indented with a tab`" because the space after the t isn’t needed unless you want both a tab and an extra space.
- Hover over the PyCharm “hints” that are on the right side of your editor to see if there are suggestions on how to make your code better.
- Can you find the code in BitBucket?
- Have you copied the URL and turned it in to Scholar? (Find the Lab 01 link on Scholar.)

Congratulations, you are done!

2.2 Lab 2: Draw a Picture

Your assignment: Draw a pretty picture. The goal of this lab is to get practice using functions to draw, and introduce computer graphics.

2.2.1 Requirements

To get full credit:

- Use the same project and repository that you used for Lab 01. If you forked your repository from mine, you should already have a folder for lab 2. Use that folder, and the `lab_02.py` file inside. Otherwise create one.
- You must use multiple colors.
- You must have a coherent picture. I am not interested in abstract art with random shapes.
- You must use multiple types of graphic functions (e.g. circles, rectangles, lines, etc.)
- Use blank lines in the code to break up sections. For example, when drawing a tree, put a blank line ahead of, and after.
- Use comments effectively. For example, when drawing a tree, put a comment at the start of those drawing commands that says `# Draw a tree`. Remember to put one space after the `#` sign.
- Put spaces after commas for proper “style.”

2.2.2 Tips

To select new colors use:

<http://www.colorpicker.com/>

Copy the values for Red, Green, and Blue. Do not worry about colors for hue, Saturation, or Brilliance.

Please use comments and blank lines to make it easy to follow your program. If you have 5 lines that draw a robot, group them together with blank lines above and below. Then add a comment at the top telling the reader what you are drawing.

Keep in mind the order of code. Shapes drawn first will be at the “back.” Shapes drawn later will be drawn on top of the other shapes.

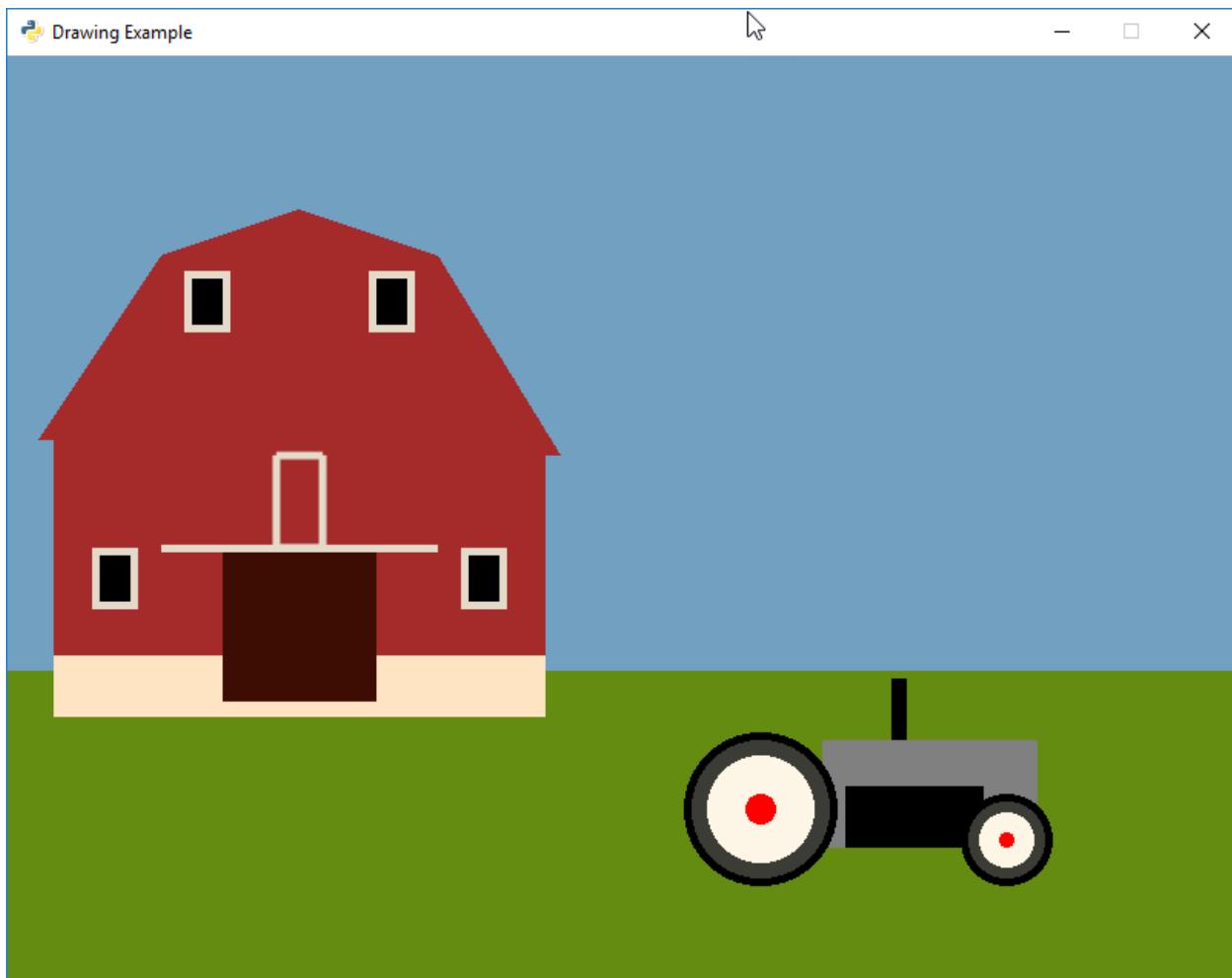
Looking for ideas? At the page below, each time you refresh the page I show various images created by students from prior years:

<http://programarcadegames.com/>

Also, remember you can look up the available commands, called the “API” at:

<http://arcade.academy>

Also, here is an example program that shows you what I’m looking for:



```
1  """
2  This is a sample program to show how to draw using the Python programming
3  language and the Arcade library.
4  """
5
6  # Import the "arcade" library
7 import arcade
8
9 # Open up a window.
10 # From the "arcade" library, use a function called "open_window"
11 # Set the window title to "Drawing Example"
12 # Set the and dimensions (width and height)
13 arcade.open_window(800, 600, "Drawing Example")
14
15 # Set the background color
16 arcade.set_background_color(arcade.color.AIR_SUPERIORITY_BLUE)
17
18 # Get ready to draw
19 arcade.start_render()
20
21 # Draw the grass
22 arcade.draw_lrtb_rectangle_filled(0, 800, 200, 0, arcade.color.BITTER_LIME)
23
24 # --- Draw the barn ---
25
26 # Barn cement base
27 arcade.draw_lrtb_rectangle_filled(30, 350, 210, 170, arcade.color.BISQUE)
28
29 # Bottom half
30 arcade.draw_lrtb_rectangle_filled(30, 350, 350, 210, arcade.color.BROWN)
31
32 # Left-bottom window
33 arcade.draw_rectangle_filled(70, 260, 30, 40, arcade.color.BONE)
34 arcade.draw_rectangle_filled(70, 260, 20, 30, arcade.color.BLACK)
35
36 # Right-bottom window
37 arcade.draw_rectangle_filled(310, 260, 30, 40, arcade.color.BONE)
38 arcade.draw_rectangle_filled(310, 260, 20, 30, arcade.color.BLACK)
39
40 # Barn door
41 arcade.draw_rectangle_filled(190, 230, 100, 100, arcade.color.BLACK_BEAN)
42
43 # Rail above the door
44 arcade.draw_rectangle_filled(190, 280, 180, 5, arcade.color.BONE)
45
46 # Draw second level of barn
47 arcade.draw_polygon_filled([[20, 350],
48                             [100, 470],
49                             [280, 470],
50                             [360, 340]]],
51                             arcade.color.BROWN)
52
53 # Draw loft of barn
54 arcade.draw_triangle_filled(100, 470, 280, 470, 190, 500, arcade.color.BROWN)
55
56 # Left-top window
57 arcade.draw_rectangle_filled(130, 440, 30, 40, arcade.color.BONE)
```

(continues on next page)

(continued from previous page)

```

58 arcade.draw_rectangle_filled(130, 440, 20, 30, arcade.color.BLACK)
59
60 # Right-top window
61 arcade.draw_rectangle_filled(250, 440, 30, 40, arcade.color.BONE)
62 arcade.draw_rectangle_filled(250, 440, 20, 30, arcade.color.BLACK)
63
64 # Draw 2nd level door
65 arcade.draw_rectangle_outline(190, 310, 30, 60, arcade.color.BONE, 5)
66
67 # --- Draw the tractor ---
68
69 # Draw the engine
70 arcade.draw_rectangle_filled(600, 120, 140, 70, arcade.color.GRAY)
71 arcade.draw_rectangle_filled(590, 105, 90, 40, arcade.color.BLACK)
72
73 # Draw the smoke stack
74 arcade.draw_rectangle_filled(580, 175, 10, 40, arcade.color.BLACK)
75
76 # Back wheel
77 arcade.draw_circle_filled(490, 110, 50, arcade.color.BLACK)
78 arcade.draw_circle_filled(490, 110, 45, arcade.color.BLACK_OLIVE)
79 arcade.draw_circle_filled(490, 110, 35, arcade.color.OLD_LACE)
80 arcade.draw_circle_filled(490, 110, 10, arcade.color.RED)
81
82 # Front wheel
83 arcade.draw_circle_filled(650, 90, 30, arcade.color.BLACK)
84 arcade.draw_circle_filled(650, 90, 25, arcade.color.BLACK_OLIVE)
85 arcade.draw_circle_filled(650, 90, 18, arcade.color.OLD_LACE)
86 arcade.draw_circle_filled(650, 90, 5, arcade.color.RED)
87
88 # --- Finish drawing ---
89 arcade.finish_render()
90
91 # Keep the window up until someone closes it.
92 arcade.run()

```

Here are some images from prior years:

To flip through the images, click the prev/next buttons above the image. The prev/next buttons below navigate between the different labs.

2.2.3 Turn In

Refer back to *Quick Reference* for a reminder on how to turn in this lab.

2.3 Lab 3: Drawing with Functions

2.3.1 Requirements

Your goal: Draw and animate an image.

Requirements for Drawing

You can update your program from Lab 2, or create a new program. This lab is worth 20 points. See the point breakdown below.

Incorporate the following items:

- Find the folder for Lab 03 in PyCharm and start entering your code there. Feel free to use any code from Lab 02 you want, just copy it across.
- Make sure you are putting your program in the Lab 3 folder, and not just changing Lab 2 to have Lab 3 requirements.
- We are going to be following the instructions/example in [*Drawing With Functions*](#).
- Put everything into a function as shown in [*Make The main Function*](#).
- Create three functions that draw something. (15 pts total, up to 5 pts per function)
 - Define the function and successfully call it. (1 pt)
 - Make your drawing function complex. 0 points for a one-line function that just draws a rectangle, 0 points for copying the example from the book, 2 points for a cohesive multi-line function. (2 pts)
 - Pass in x and y parameters and successfully position the object as shown in [*Make The Drawing Functions*](#). (2 pts)

Drawing with functions is worth 16 points. You can get the final 4 points by animating your image.

Requirements for Animation

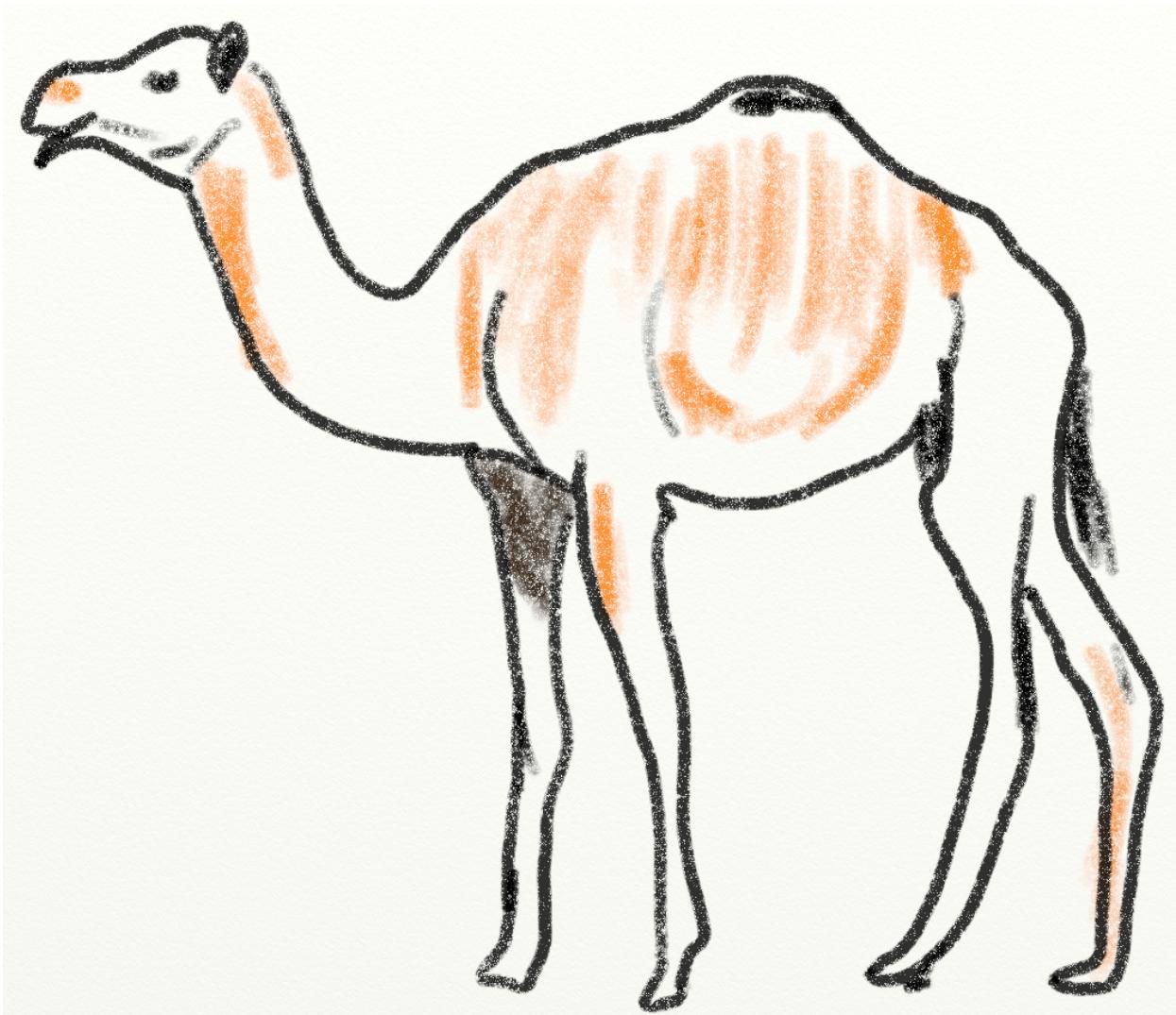
Animate an object. The movement does not need to be complex. See [*How To Animate A Drawing Function*](#) for an example.

Double Check

Make sure you don't put functions inside of functions. After the `import` statement, each function should be listed out, one `def` after the other. But no `def` inside of another `def`.

Also make sure you have *three* functions that take in an `(x, y)` position to draw an object, not just one.

2.4 Lab 4: Camel



2.4.1 Description of the Camel Game

The idea for Camel originally came from the Heath Users Group and was published in [More BASIC Computer Games](#) in 1979.

The idea is to ride your camel across the desert while being chased. You need to manage your thirst, how tired the camel is, and how far ahead of the natives you are.

This is one of the first games I programmed on the Apple //e. The game is flexible. I've had students create Star Wars themed versions of this game where you need to ride a wampa across Hoth. It is easy to add sandstorms and other random events to the game to make it more interesting. Even though it is only a text-based game, I've found some students really get into playing it. Once I had a professor come and complain when my students were too loud while playing this game on the "big screen."

2.4.2 Sample Run of Camel

Here is a sample run of the game:

Welcome to Camel!

You have stolen a camel to make your way across the great Mobi desert. The natives want their camel back and are chasing you down! Survive your desert trek and outrun the natives.

- A. Drink from your canteen.
 - B. Ahead moderate speed.
 - C. Ahead full speed.
 - D. Stop and rest.
 - E. Status check.
 - Q. Quit.
- Your choice? C

You traveled 12 miles.

- A. Drink from your canteen.
 - B. Ahead moderate speed.
 - C. Ahead full speed.
 - D. Stop and rest.
 - E. Status check.
 - Q. Quit.
- Your choice? C

You traveled 17 miles.

- A. Drink from your canteen.
 - B. Ahead moderate speed.
 - C. Ahead full speed.
 - D. Stop and rest.
 - E. Status check.
 - Q. Quit.
- Your choice? e

Miles traveled: 29

Drinks in canteen: 3

The natives are 31 miles behind you.

- A. Drink from your canteen.
 - B. Ahead moderate speed.
 - C. Ahead full speed.
 - D. Stop and rest.
 - E. Status check.
 - Q. Quit.
- Your choice? b

You traveled 6 miles.

...and so on until...

- A. Drink from your canteen.
- B. Ahead moderate speed.
- C. Ahead full speed.
- D. Stop and rest.
- E. Status check.

(continues on next page)

(continued from previous page)

```

Q. Quit.
Your choice? C

You traveled 12 miles.
The natives are getting close!

A. Drink from your canteen.
B. Ahead moderate speed.
C. Ahead full speed.
D. Stop and rest.
E. Status check.
Q. Quit.
Your choice? C

You traveled 11 miles.
The natives are getting close!
You made it across the desert! You won!

```

This game assumes you know the material up through *Guessing Games with Random Numbers and Loops*.

2.4.3 Programming Guide

Here are the steps to complete this lab. Feel free to modify and add to the lab. Try the game with friends and family.

1. Create a new directory in your project for Lab 04 – Camel and a file called lab_04.py in that folder if the folder and file don't exist already.
2. In that file create a new main() function. Have it print the instructions to the screen. Do this with multiple print statements. Don't use one print statement and multiple \n characters to jam everything on one line.

```

Welcome to Camel!
You have stolen a camel to make your way across the great Mobi desert.
The natives want their camel back and are chasing you down! Survive your
desert trek and out run the natives.

```

Call the main() function. Run and test the program.

3. Continue from the prior step and create a Boolean variable called done and set to False. Make sure this, and everything else, is in the main function.
4. Create a while loop that will keep looping while done is False.
5. Inside the loop, print out the following:

```

A. Drink from your canteen.
B. Ahead moderate speed.
C. Ahead full speed.
D. Stop for the night.
E. Status check.
Q. Quit.

```

6. Ask the user for their choice. Make sure to add a space before the quote so the user input doesn't run into your text. That is, it should look like:

```
What is your choice? Q
```

And not:

What is your choice?Q

7. If the user's choice is Q, then set done to True. By doing something like `user_choice.upper()` instead of just `user_choice` in your `if` statement you can make it case-insensitive. Also, print something to the user to let her know she quit the game. Users need feedback.
8. Test and make sure that you can quit out of the game, and that it works for both upper and lower case.
9. Before your main program loop, create variables for miles traveled, thirst, camel tiredness. Set these to zero. Creating accurate, descriptive variable names will make the rest of this lab less confusing.
10. Create a variable for the distance the natives have traveled and set it to -20. (Twenty miles back.) Understanding how we track the player's distance, the native's distance, and the difference between the two tends to confuse some people. See the videos at the end of this lab for some hints.
11. Create and set an initial number of drinks in the canteen. I use three.
12. Go back to inside your main program loop. Add an `elif` to the `if` that is checking for the quit command. See if the user is asking for status. If so, print out the miles traveled, the drinks in the canteen, and how far the natives are behind you. If you aren't sure how to calculate that, see the videos at the end of this lab. (Seriously. This is a section where a lot of people start down the wrong path.)

```
Miles traveled:  0  
Drinks in canteen:  3  
The natives are 10 miles behind you.
```

13. Add an `elif` in your main program loop and handle if the user wants to stop for the night. If the user does, reset the camel's tiredness to zero. Print that the camel is happy, and move the natives up a random amount from 7 to 14 or so. Note: calculate random numbers IN the loop, just before you need them. A common mistake is to calculate the random numbers you need BEFORE the main loop. Causing your random numbers to stay the same the entire game.
14. Add an `elif` in your main program loop and handle if the user wants to go ahead full speed. If the user does, go forward a random amount between 10 and 20 inclusive. Print how many miles the user traveled. Add 1 to thirst. Add a random 1 to 3 to camel tiredness. Move the natives up 7 to 14 miles.
15. Add an `elif` in your main program loop and handle if the user wants to go ahead moderate speed. If the user does, go forward a random amount between 5 and 12 inclusive. Print how many miles the user traveled. Add 1 to thirst. Add 1 to camel tiredness. Move the natives up 7 to 14 miles.
16. Add an `elif` in your main program loop and handle if the user wants to go ahead drink from the canteen. If the user does, make sure there are drinks in the canteen. If there are, subtract one drink and set the player's thirst to zero. Otherwise print an error.
17. In the loop, print "You are thirsty." if the user's thirst is above 4. Put this after your long `if ... elif` chain above.
18. Print "You died of thirst!" if the user's thirst is above 6. Set `done` to True. Make sure you create your code so that the program doesn't print both "Your are thirsty" and "You died of thirst!" Use `elif` as appropriate.
19. Print "Your camel is getting tired." if the camel's tiredness is above 5.
20. Print "Your camel is dead." if the camel's tiredness is above 8. Like the prior steps, print one or the other. It is a good idea to include a check with the `done` variable so that you don't print that your camel is getting tired after you died of thirst.
21. If the natives have caught up, print that they caught the player and end the game.
22. Else if the natives are less than 15 miles behind, print "The natives are getting close!"

23. If the user has traveled 200 miles across the desert, print that they won and end the game. Make sure they aren't dead before declaring them a winner. If they land on mile marker 201 instead of 200, make sure they still win the game. See the videos below.
24. Add a one-in-twenty chance of finding an oasis. Print that the user found it, refill the canteen, reset player thirst, and rest the camel. If you aren't sure how to do this, see [Random Chance](#). Make sure a person can't find the oasis unless they are traveling. This will require you to think about where you are putting this code.
25. Play the game and tune the numbers so it is challenging but not impossible. Fix any bugs you find.

2.4.4 Hints

- Remember that it is good idea to put blank lines between logical groupings of code in your program. For example, but a blank line after the instructions, and between each user command. Add comments where appropriate.
- It is considered better style to use `while not done:` instead of `while done == False:`
- Rather than setting `done` to `True`, you can immediately exit the main program loop by using the `break` statement. Some people prefer this method because it simplifies the logic. Other people prefer not jumping out of the middle of a loop. Quite frankly, either works.
- To prevent bad message combinations, such as printing "You died of thirst." and "You found an oasis!" on the same turn, use the `and` operator. Such as, `if not done and thirst > 4:`
- Make sure you add blank lines to divide up sections of code. Don't double space the code. Add comments ahead of code sections.
- Watch the English and punctuation. Don't say `you found an Oasis!`, say `You found an oasis!`
- Occasionally people get confused about how to set a value back to zero. Study the code below:

```
# Yes, these set thirst to zero, but this is NOT the way to do it.
thirst *= 0          # No
thirst = thirst * 0    # No
thirst = thirst - thirst  # No

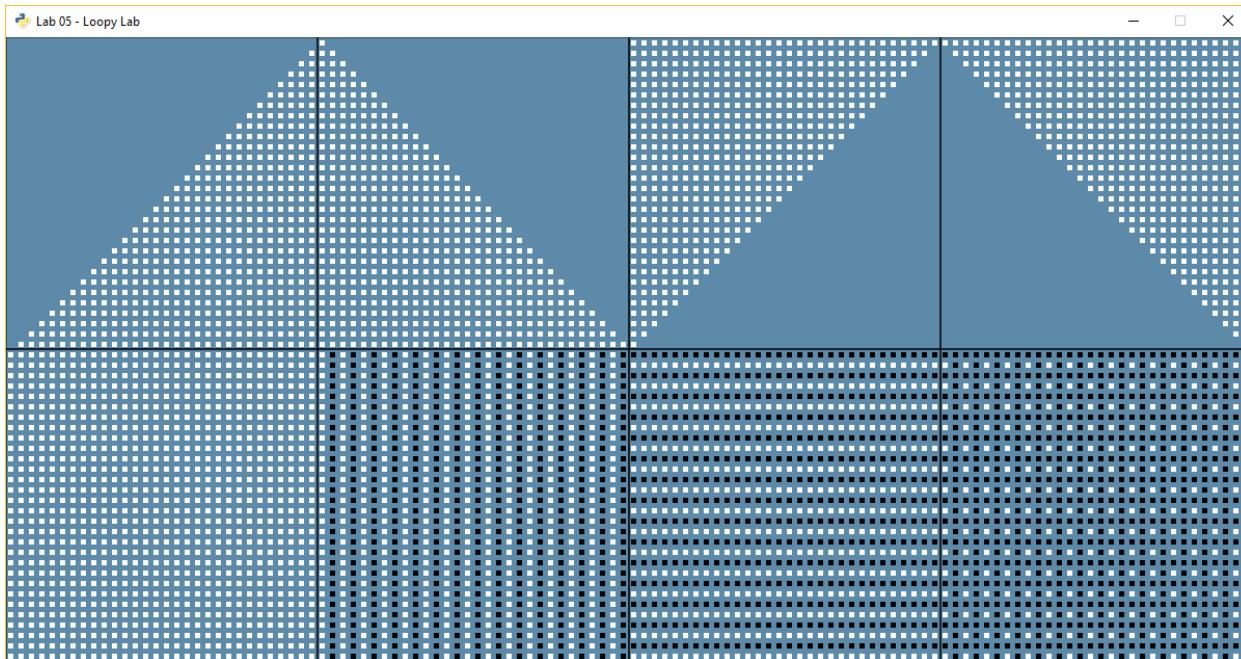
# This is the correct way to set thirst to zero
thirst = 0          # Yes
```

Calculating How Far Back the Natives Are

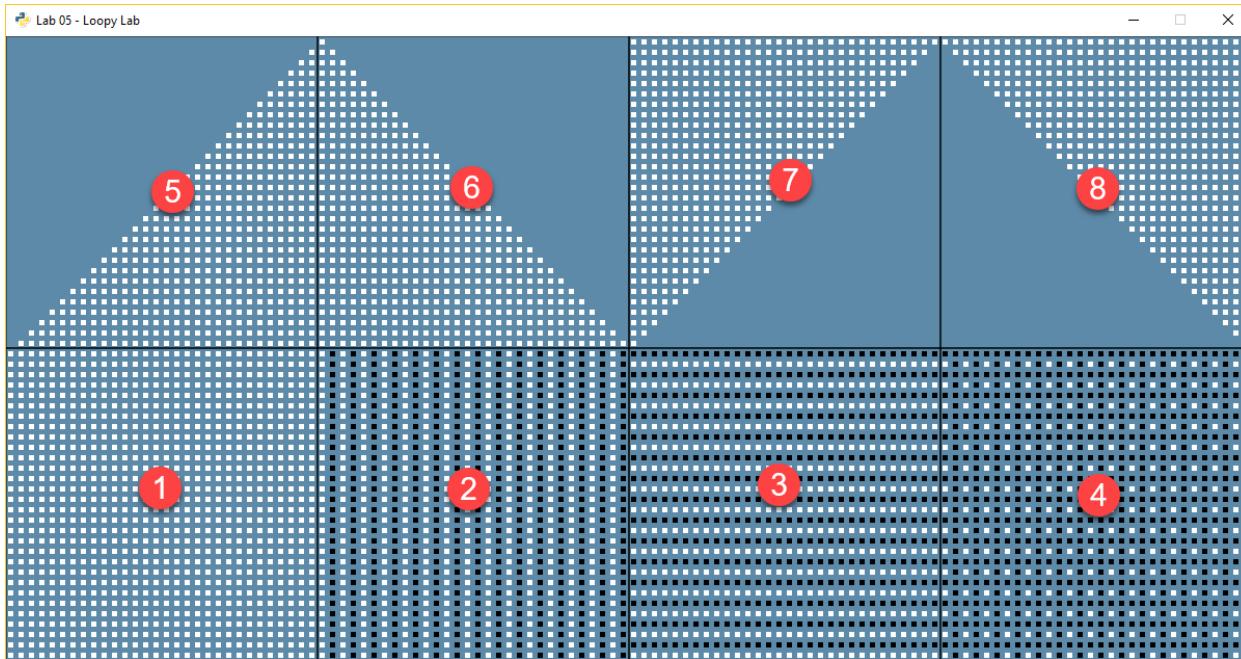
Figuring Out The End of the Game

2.5 Lab 5: Loopy Lab

The goal of this lab is to practice using nested `for` loops. We will create a program that makes the following image:



For sanity's sake, let's give a number to each section. Then when we ask questions we can say which section we are talking about. **Do not** put the numbers in the final program. Here are the section numbers:



Great! All these items can be created using nested `for` loops.

2.5.1 Getting Started

Below is some code to get you started. Underneath each of the comments, fill in the code required to make the pattern.

```
import arcade
```

(continues on next page)

(continued from previous page)

```

def draw_section_outlines():
    # Draw squares on bottom
    arcade.draw_rectangle_outline(150, 150, 300, 300, arcade.color.BLACK)
    arcade.draw_rectangle_outline(450, 150, 300, 300, arcade.color.BLACK)
    arcade.draw_rectangle_outline(750, 150, 300, 300, arcade.color.BLACK)
    arcade.draw_rectangle_outline(1050, 150, 300, 300, arcade.color.BLACK)

    # Draw squares on top
    arcade.draw_rectangle_outline(150, 450, 300, 300, arcade.color.BLACK)
    arcade.draw_rectangle_outline(450, 450, 300, 300, arcade.color.BLACK)
    arcade.draw_rectangle_outline(750, 450, 300, 300, arcade.color.BLACK)
    arcade.draw_rectangle_outline(1050, 450, 300, 300, arcade.color.BLACK)

def draw_section_1():
    for row in range(30):
        for column in range(30):
            x = 0 # Instead of zero, calculate the proper x location using 'column'
            y = 0 # Instead of zero, calculate the proper y location using 'row'
            arcade.draw_rectangle_filled(x, y, 5, 5, arcade.color.WHITE)

def draw_section_2():
    # Below, replace "pass" with your code for the loop.
    # Use the modulus operator and an if statement to select the color
    # Don't loop from 30 to 60 to shift everything over, just add 300 to x.
    pass

def draw_section_3():
    # Use the modulus operator and an if/else statement to select the color.
    # Don't use multiple 'if' statements.
    pass

def draw_section_4():
    # Use the modulus operator and just one 'if' statement to select the color.
    pass

def draw_section_5():
    # Do NOT use 'if' statements to complete 5-8. Manipulate the loops instead.
    pass

def draw_section_6():
    pass

def draw_section_7():
    pass

def draw_section_8():
    pass

```

(continues on next page)

(continued from previous page)

```
def main():
    # Create a window
    arcade.open_window(1200, 600, "Lab 05 - Loopy Lab")
    arcade.set_background_color(arcade.color.AIR_FORCE_BLUE)

    arcade.start_render()

    # Draw the outlines for the sections
    draw_section_outlines()

    # Draw the sections
    draw_section_1()
    draw_section_2()
    draw_section_3()
    draw_section_4()
    draw_section_5()
    draw_section_6()
    draw_section_7()
    draw_section_8()

    arcade.finish_render()

    arcade.run()

main()
```

2.5.2 Scoring

There are 20 possible points:

- Section 1: 2 pts
- Section 2: 2 pts
- Section 3: 2 pts
- Section 4: 2 pts
- Completing any section 1-4: 1 pt
- Section 5: 2 pts
- Section 6: 2 pts
- Section 7: 2 pts
- Section 8: 2 pts
- Completing any section 5-8: 1 pt
- Code Style: 2 pts (Should have no yellow lines on right side of PyCharm)

2.5.3 Hints

- Each little square is a 5x5 pixel square.
- If the center of each square is 5 apart, you won't see an edge and it will look like one big square.

- Remember, row controls up and down, so it corresponds to y. Column corresponds to x.
- When working on sections 2-8, you can simple add to the x and/or y values to shift everything over. For example, just add 300 to all the x values in section 2, to put it in the second box.
- Each section is 300x300 pixels.
- You will only need two `for` loops. Do not use a third nested `for` loop.
- Section 4 only needs one `if` statement. The trick is to use an `and`.
- Remember that the `%` sign is the modulus. It calculates the remainder. So:

```
0 % 3 = 0
1 % 3 = 1
2 % 3 = 2
3 % 3 = 0
4 % 3 = 1
5 % 3 = 2
6 % 3 = 0
```

- You can count “backwards” by subtracting.

```
print("Count up!")
for i in range(10):
    print(i)

print()
print("Count down, even if i is going up")
for i in range(10):
    x = 9 - i
    print("i is", i, "and 9-i is", x)
```

This prints:

```
Count up!
0
1
2
3
4
5
6
7
8
9

Count down, even if i is going up
i is 0 and 9-i is 9
i is 1 and 9-i is 8
i is 2 and 9-i is 7
i is 3 and 9-i is 6
i is 4 and 9-i is 5
i is 5 and 9-i is 4
i is 6 and 9-i is 3
i is 7 and 9-i is 2
i is 8 and 9-i is 1
i is 9 and 9-i is 0
```

2.6 Lab 6: Text Adventure



2.6.1 Description of the Adventure Game

One of the first games I ever played was a text adventure called [Colossal Cave Adventure](#). You can play the game on-line [here](#) to get an idea what text adventure games are like. Seriously, give it a try. Otherwise it will be hard to understand what we are trying to do here.

Arguably the most famous of this genre of game is the [Zork](#) series.

The first “large” program I created myself was a text adventure. It is easy to start an adventure like this. It is also a great way to practice using lists. Our game for this lab will involve a list of rooms that can be navigated by going north, east, south, or west. Each room will be a list with the room description, and then what rooms are in each of the directions. See the section below for a sample run:

2.6.2 Sample Run

```
You are in a dusty castle room.  
Passages lead to the north and south.  
What direction? n  
  
You are in the armory.  
There is a room off to the south.  
What direction? s
```

(continues on next page)

(continued from previous page)

You are in a dusty castle room.
Passages lead to the north and south.
What direction? s

You are in a torch-lit hallway.
There are rooms to the east and west.
What direction? e

You are in a bedroom. A window overlooks the castle courtyard.
A hallway is to the west.
What direction? w

You are in a torch-lit hallway.
There are rooms to the east and west.
What direction? w

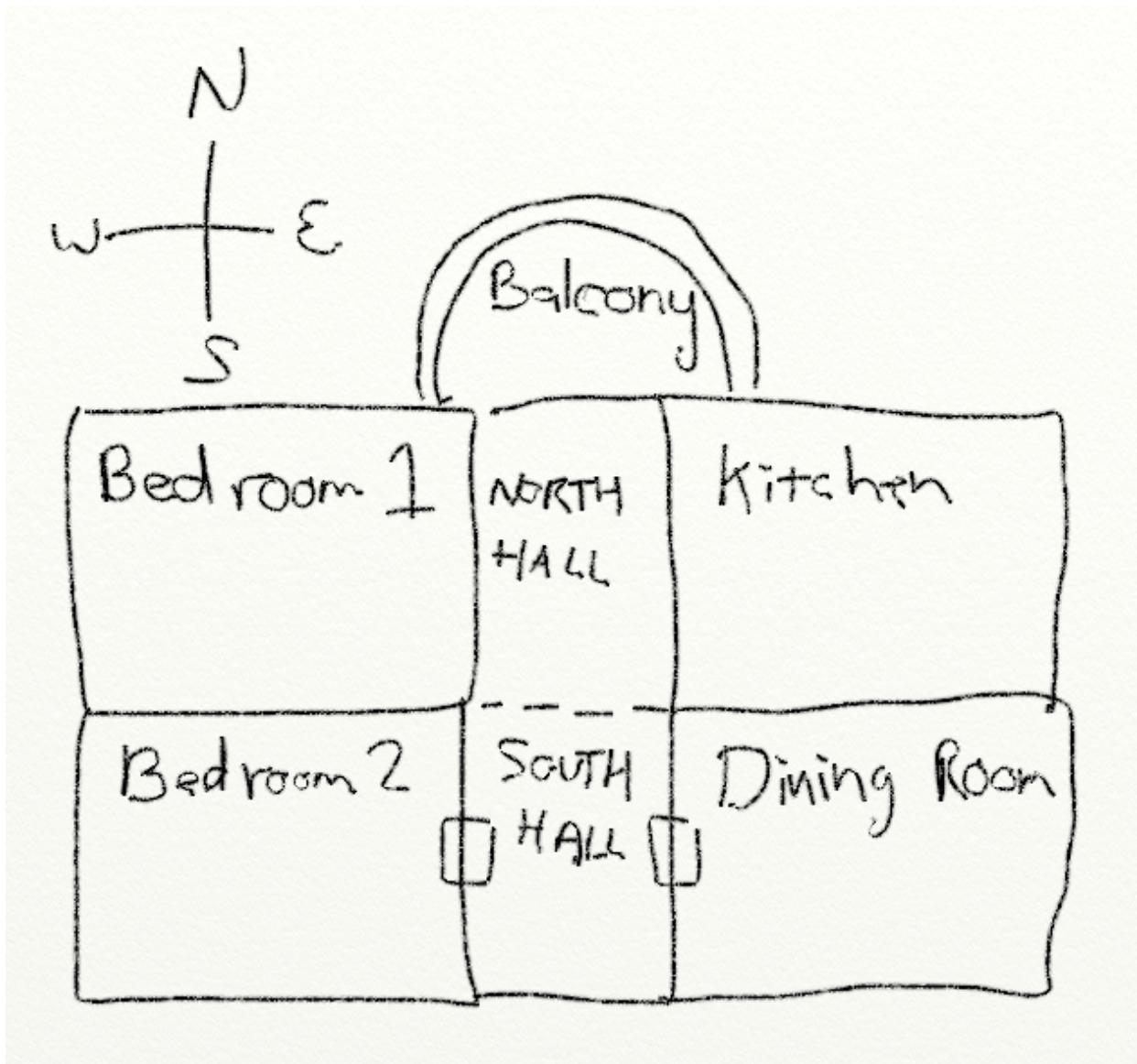
You are in the kitchen. It looks like a roast is being made for supper.
A hallway is to the east.
What direction? w

Can't go that way.
You are in the kitchen. It looks like a roast is being made for supper.
A hallway is to the east.
What direction?

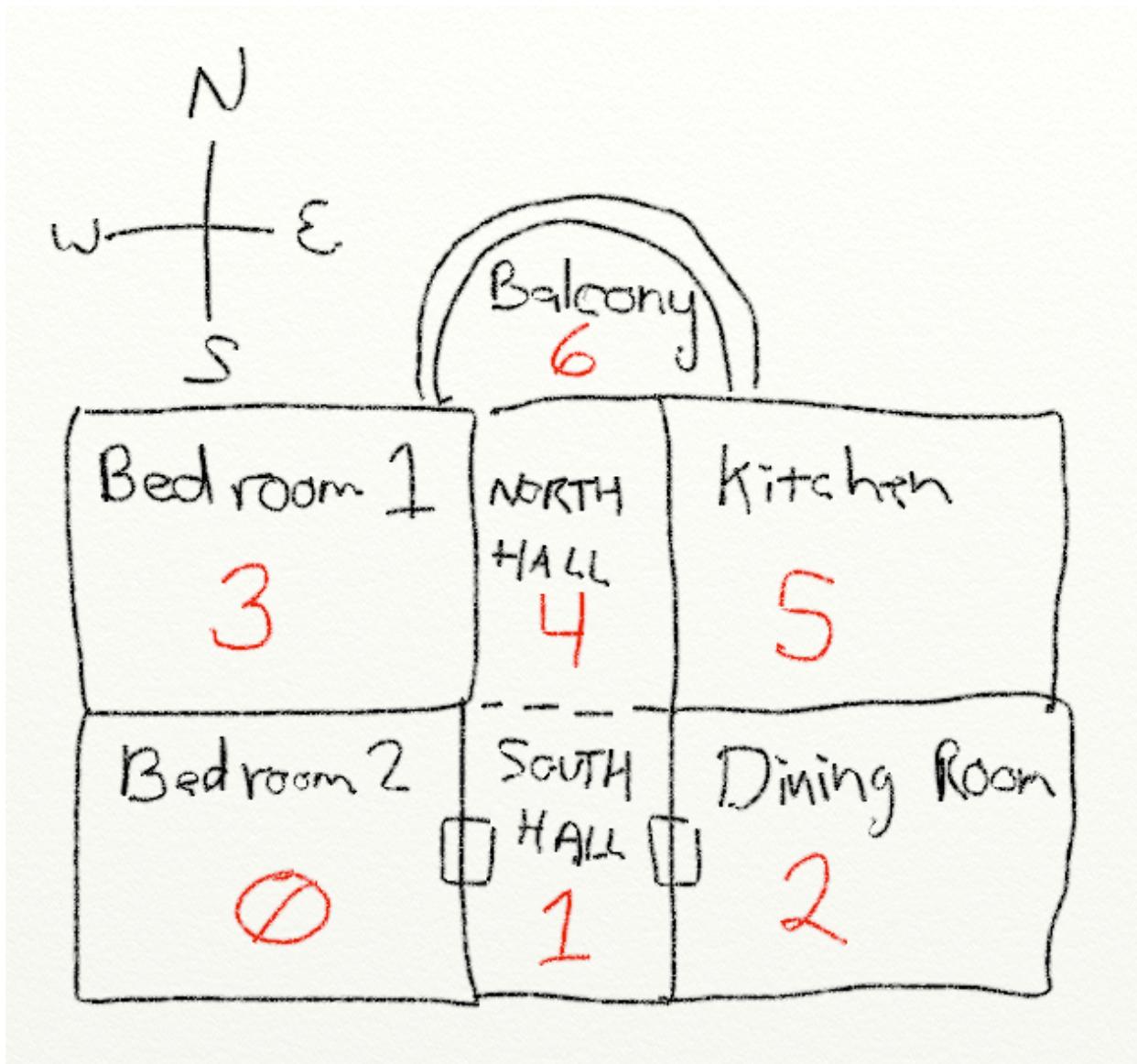
This game assumes you know the material up through *Introduction to Lists*.

2.6.3 Creating Your Dungeon

Before you start, sketch out the dungeon that you want to create. It might look something like this:



Next, number all of the rooms starting at zero.



Use this sketch to figure out how all the rooms connect. For example, room 0 connects to room 3 to the north, room 1 to the east, and no room to the south and west.

2.6.4 Step-by-step Instructions

As a reminder, at the end of the year I do scan for duplicate homework. I keep all homework assignments from prior semesters, and assignments from non-Simpson students that I find on-line. I run a program that scans for duplicates. Make sure your work is your own.

1. Create a `main` function and call the `main` function.
2. In the `main` function, create an empty array called `room_list`. If you've forgotten, see [Create an Empty List](#).
3. Create a variable called `room`. Set it equal to an array with five elements. For the first element, create a string with a description of your first room. The last four elements will be the number of the next room if the user goes north, east, south, or west. Look at your sketch to see what numbers to use. Use `None` if no room hooks up in that direction. (Do not put `None` in quotes. Also, remember that Python is case sensitive so `none` won't work)

either. The keyword `None` is a special value that represents “nothing.” Because sometimes you need a value, other than zero, that represents)

4. Append this room to the room list. See [Adding to a List](#) if you’ve forgotten how to do that.
5. Repeat the prior two steps for each room you want to create. Just re-use the room variable.
6. Create a variable called `current_room`. Set it to zero.
7. Print the `room_list` variable. Run the program. You should see a really long list of every room in your adventure. If you don’t, make sure you are calling your `main` function at the end of your program, and that it isn’t indented.
8. Adjust your print statement to only print the first room (element zero) in the list. Note that at index 0 is the description, 1 is the room to the north, 2 is the room to the east, etc. Run the program and confirm you get output similar to:

```
[ 'You are in a room. There is a passage to the north.', 1, None, None, None ]
```

9. Using `current_room` and `room_list`, print the current room the user is in. Since your first room is zero, the output should be the same as before.
10. Change the print statement so that you only print the description of the room, and not the rooms that hook up to it. Remember if you are printing a list in a list the index goes after the first index. Don’t do this: `[current_room[0]]`, do `[current_room][0]`

```
You are in a room. There is a passage to the north.
```

11. Create a variable called `done` and set it to `False`. Then put the printing of the room description in a `while` loop that repeats until `done` is set to `True`. We won’t set `done` to `True` yet though.
12. Before printing the description, add a code to print a blank line. This will make it visually separate each turn when playing the game.
13. After printing the room description, add a line of code that asks the user what they want to do. Use the `input` statement. Keep in mind that you will be entering letters, therefore you will *not* want to convert what the user enters to an integer or floating point number. This will be similar to how we got input in [Lab 4: Camel](#). The most frequent mistake I’ve seen students make is to have an `input` statement and not capture the return value. See [Capturing Returned Values](#) if you have this issue.
14. Add an `if` statement to see if the user wants to go north. You should accept user input like “n” and “N” and “North” and “NoRtH”. You may need to review [Text Comparisons](#) and [Multiple Text Possibilities](#).
15. If the user wants to go north, create a variable called `next_room` and get it equal to `room_list[current_room][1]`, which should be the number for what room is to the north. (Remember, 0 is the description, 1 is north, 2 is east, etc.)
16. Add another `if` statement to see if the next room is equal to `None`. If it is, print “You can’t go that way.” *Otherwise* (how do you do ‘otherwise’?) set `current_room` equal to `next_room`. Note: This new `if` statement is part of the `if` statement to go north. So make sure it is indented inside that `if`.
17. Test your program. Can you go north to a new room?
18. Add `elif` statements to handle east, south, and west. Add an `else` statement to let the user know the program doesn’t understand what she typed.
19. It is a great idea to put blank lines between the code that handles each direction. I don’t mean to print a blank line, but actually have blank lines in the code. That way you visually group the code into sections.
20. It is a great idea to add comments too, to each section.

21. Test your program. Make sure you have enough of a description that someone running the program will know what direction to go. Don't say "You are in the kitchen." Instead say "You are in the kitchen. There is a door to the north."
22. Add a quit command that ends the game.
23. Make sure that the program works for upper and lower case commands.
24. Have the program work if the user types in "north" or "n". Review [Multiple Text Possibilities](#) if needed.

Spend a little time to make this game interesting. Don't simply create an "East room" and a "West room." That's boring.

Also spend a little time to double check spelling and grammar. Without a word processor checking your writing, it is important to be careful. Pay particular note to:

- Students often capitalize words in this lab that should not be capitalized. In particular, see [when do you capitalize directions](#).
- Do not capitalize room names unless the room name is part of a title. Don't say "You are in the Living Room," because the word "living room" isn't normally capitalized.

Use \n to add carriage returns in your descriptions so they don't print all on one line. Don't put spaces around the \n, or the spaces will print.

What I like about this program is how easy it is to expand into a full game. Using all eight cardinal directions (including "NorthWest"), along with "up" and "down" is rather easy. Managing an inventory of objects that can exist in rooms, be picked up, and dropped is also a matter of keeping lists.

Expanding this program into a full game is one of the two options for the final lab in this course.

2.7 Lab 7: User Control

This lab gives you a chance to practice drawing an object with a function and allow the user to control it.

Attention: This lab does **NOT** use sprites. Look at [User Control](#) for examples relevant to this lab.

Write one program, following these steps:

- Create a Lab 07 – User Control folder for this lab, or use one that is already set up in your project.
- Begin your code with the template below:

Listing 1: Lab 7 Template

```

1  """ Lab 7 - User Control """
2
3 import arcade
4
5 # --- Constants ---
6 SCREEN_WIDTH = 800
7 SCREEN_HEIGHT = 600
8
9
10 class MyGame(arcade.Window):
11     """ Our Custom Window Class"""
12
13     def __init__(self):

```

(continues on next page)

(continued from previous page)

```

14     """ Initializer """
15
16     # Call the parent class initializer
17     super().__init__(SCREEN_WIDTH, SCREEN_HEIGHT, "Lab 7 - User Control")
18
19     def on_draw(self):
20         arcade.start_render()
21
22
23 def main():
24     window = MyGame()
25     arcade.run()
26
27
28 main()

```

- Create a background image of some sort in the `on_draw` method. Feel free to use any of *your* code that *you* have written from prior labs that you have. To do this, put your code in the `on_draw` method. If your code used functions, move those functions as well to this program. Paste them above the `MyGame` class. Don't turn this in with just a solid color for the background, or you'll lose a couple points.
- In *User Control*, we talked about moving graphics with the keyboard, a game controller, and the mouse. To start, pick one of these methods, and get that ball moving around in your program.
- In the examples, we just moved a ball. Adjust the code so that it draws something more interesting than a ball. You can pull from the code you used in prior labs where you drew with a function, and move it to the `on_draw` method of your class. But don't have your object be a simple one line of code shape.
- Rename your class from `Ball` to a name that represents what you are really drawing.
- After you get that object moving, pick a *different* method of control (keyboard, mouse, gamepad.) Create a *new class*, draw something *different*, and move it around the screen.
- In the case of the game controller and the keyboard, make sure to add checks so that your object does not move off-screen and get lost.
- Add a sound effect for when the user clicks the mouse button.
- Add a different sound effect for when the user bumps into the edge of the screen. Make sure the sound only plays when you are trying to go beyond the edge of the screen, and not when you just sit at the edge of the screen. This part isn't needed for anything controlled by the mouse.

When you are done, make sure you commit your sound effects in what you turn in! Otherwise your program can't be graded.

2.8 Lab 8: Sprites

- Create a player-character sprite. Feel free to pick your own image for the sprite.
- Allow the user to move the player move by the keyboard, mouse, or game pad. Your choice. If you use the mouse, make sure you hide it.
- Create “good” sprites. Pick your own image for the sprite.
- Create “bad” sprites. Pick your own image for the sprite.
- For each good sprite collected, make the score go up. Play a ‘good’ sound when the user collects that item.
- For each bad sprite touched, have the score go down. Play a ‘bad’ sound when the user collects that item.

- Animate/move the good sprites. Pick any of the motions we showed, or make your one.
- Move the bad sprites. Move them differently than the good ones.
- If the length of the good sprite list is zero, then don't move any of the sprites or the character. "Freeze" the game. To do this, note what line of the `update` method in the main window class causes the sprites to move. Then only run that line if you have sprites. (Check the length of the sprite list.) If you get that to work, the do something similar on `_mouse_motion`.
- If the length of the good sprite list is zero, draw "Game Over" on the screen.
- Add, commit, and push via git.
- Turn in a URL to your project.

2.9 Lab 9: Sprites and Walls

Goal: Create a landscape of wall objects that the user must navigate around to collect coins. This will help practice using `for` loops to create and position multiple items.

- Step 1: Start with one of the move with walls examples. Use either the [Move with Walls](#) example or the [Move with a Scrolling Screen](#) example.
- Step 2: If you start with the examples, delete the current wall placement code. You want to create your own.
- Step 3: Create a more complex arrangement of walls. Make sure the walls don't allow the user to go off-screen. 6 points, based on how complex the arrangement. See [Individually Placing Walls](#), [Placing Walls With A Loop](#), and [Placing Walls With A List](#) for ideas. Just DON'T do the same thing as examples. Make it your own. Also, if you have more than one type of wall block, that's great. But you don't need more than one wall list. The physics engine requires all walls be kept in the same list.
- Step 4: Update the graphics. Use multiple types of blocks for the walls. Maybe change the character. 4 points, one point for each graphic used that wasn't part of the base example. Remember to put a quick citation in your program just before you load graphics or sounds.
- Step 5: Add coins (or something) for the user to collect. 4 points, based on the complexity of the coin layout. Remember, you can place coins like we placed wall blocks. If you randomly place coins, you might end up with coins on top of walls. See the "Important Part" around line 75 or so of the example [sprite_no_coins_on_walls.py](#) for how to avoid this.
- Step 6: Keep score of how many coins were collected, and display on-screen. 4 points.
- Step 7: Add a sound to play each time the user collects a coin. 2 points.

Warning: Don't move the player twice!

The command `self.physics_engine.update()` moves the player while checking for walls. The command `self.all_sprites_list.update()` will move the player WITHOUT checking for walls. Don't do both commands. You'll end up "walking through walls." If you have other sprites to update, update only those sprites. For example: `self.coin_list.update()`.

2.9.1 Additional Challenges

These aren't required for the lab, but I've had students ask in prior years how to do these:

- If you are interested in having the player be able to face left or right, see the [Sprite Face Left or Right](#) example.

- Want to animate walking? Look at the [Animate your sprites.](#) example.

2.10 Lab 10: Spell Check

This lab shows how to create a spell checker. To prepare for the lab, download the files listed below.

- [AliceInWonderLand200.txt](#) - First chapter of “Alice In Wonderland”
- [dictionary.txt](#) - A list of words

2.10.1 Requirements

Write a single program in Python that checks the spelling of the first chapter of “Alice In Wonderland.” First use a linear search, then use a binary search. Print the line number along with the word that does not exist in the dictionary.

Follow the steps below carefully. If you don’t know how to accomplish one step, ask before moving on to the next step.

2.10.2 Steps to complete

1. Find or create a directory for your project.
2. Download the dictionary to the directory.
3. Download first 200 lines of Alice In Wonderland to your directory.
4. Start a Python file for your project.
5. It is necessary to split apart the words in the story so that they may be checked individually. It is also necessary to remove extra punctuation and white-space. Unfortunately, there is not any good way of doing this with what the book has covered so far. The code to do this is short, but a full explanation is beyond the scope of this class. Include the following function in your program. Remember, function definitions should go at the top of your program just after the imports. We’ll call this function in a later step.

Listing 2: Function to split apart words in a string and return them as a list

```
import re

# This function takes in a line of text and returns
# a list of words in the line.
def split_line(line):
    return re.findall('[A-Za-z]+(?:\'[A-Za-z]+)?', line)
```

This code uses a *regular expression* to split the text apart. Regular expressions are very powerful and relatively easy to learn. If you’d like to know more about regular expressions, see:

<http://regexone.com/>

6. Read the file `dictionary.txt` into an array. Go back to [Reading Into an Array](#) for example code on how to do this. This does *not* have anything to do with the `import` command, libraries, or modules. Don’t call the dictionary `word_list` or something generic because that will be confusing. Call it `dictionary_list` or a different term that is specific.
7. Close the file.
8. Print --- Linear Search ---

9. Open the file `AliceInWonderLand200.txt`
10. We are *not* going to read the story into a list. Do not create a new list here like you did with the dictionary.
11. Start a `for` loop to iterate through each line.
12. Call the `split_line` function to split apart the line of text in the story that was just read in. Store the list that the function returns in a new variable named `word_list`. Remember, just calling the function won't do anything useful. You need to assign a variable equal (`word_list`) to the result. If you've forgotten how to capture the return value from a function, see [Capturing Returned Values](#).
13. Start a nested `for` loop to iterate through each word in the words list. This should be inside the `for` loop that runs through each line in the file. (One loop for each line, another loop for each word in the line.)
14. Using a linear search, check the current word against the words in the dictionary. See [Linear Search Algorithm](#) for example code on how to do this. The linear search is just three lines long. When comparing to the word to the other words in the dictionary, convert the word to uppercase. In your `while` loop just use `word.upper()` instead of `word` for the key. This linear search will exist inside the `for` loop created in the prior step. We are looping through each word in the dictionary, looking for the current word in the line that we just read in.
15. If the word was not found, print the word. Don't print anything if you do find the word, that would just be annoying.
16. Close the file.
17. Make sure the program runs successfully before moving on to the next step.
18. Create a new variable that will track the line number that you are on. Print this line number along with the misspelled from the prior step.
19. Make sure the program runs successfully before moving on to the next step.
20. Print --- Binary Search ---
21. The linear search takes quite a while to run. To temporarily disable it, it may be commented out by using three quotes before and after that block of code. Ask if you are unsure how to do this.
22. Repeat the same pattern of code as before, but this time use a binary search. For the binary search, go back to [Binary Search](#). Much of the code from the linear search may be copied, and it is only necessary to replace the lines of code that represent the linear search with the binary search.
23. Note the speed difference between the two searches.
24. Make sure the linear search is re-enabled, if it was disabled while working on the binary search.
25. Upload the final program or check in the final program.

2.10.3 Example Run

```
--- Linear Search ---
Line 3  possible misspelled word: Lewis
Line 3  possible misspelled word: Carroll
Line 46 possible misspelled word: labelled
Line 46 possible misspelled word: MARMALADE
Line 58 possible misspelled word: centre
Line 59 possible misspelled word: learnt
Line 69 possible misspelled word: Antipathies
Line 73 possible misspelled word: curtsey
Line 73 possible misspelled word: CURTSEYING
Line 79 possible misspelled word: Dinah'11
Line 80 possible misspelled word: Dinah
```

(continues on next page)

(continued from previous page)

```
Line 81 possible misspelled word: Dinah
Line 89 possible misspelled word: Dinah
Line 89 possible misspelled word: Dinah
Line 149 possible misspelled word: flavour
Line 150 possible misspelled word: toffee
Line 186 possible misspelled word: croquet
--- Binary Search ---
Line 3 possible misspelled word: Lewis
Line 3 possible misspelled word: Carroll
Line 46 possible misspelled word: labelled
Line 46 possible misspelled word: MARMALADE
Line 58 possible misspelled word: centre
Line 59 possible misspelled word: learnt
Line 69 possible misspelled word: Antipathies
Line 73 possible misspelled word: curtsey
Line 73 possible misspelled word: CURTSEYING
Line 79 possible misspelled word: Dinah'll
Line 80 possible misspelled word: Dinah
Line 81 possible misspelled word: Dinah
Line 89 possible misspelled word: Dinah
Line 89 possible misspelled word: Dinah
Line 149 possible misspelled word: flavour
Line 150 possible misspelled word: toffee
Line 186 possible misspelled word: croquet
```

2.11 Lab 11: Array-Backed Grids

2.11.1 Part 1

Start with the program at the end of *Array-Backed Grids*. Modify it so that rather than just changing the block the user clicks on, also change the blocks of the squares next to the user's click. (5 pts) If the user clicks on an edge, make sure the program doesn't crash and still handles the click appropriately. (5 pts)

Fig. 1: Example of working Lab 11

2.11.2 Part 2

Create another program, again starting with the code at the end of *Array-Backed Grids*.

Write a loop that will count all of the cells that are selected in the grid and print them out. Put this code at the end of your `on_mouse_press` function. (2 pts)

Listing 3: Sample Output

```
Total of 1 cells are selected.
Total of 2 cells are selected.
Total of 3 cells are selected.
Total of 2 cells are selected.
Total of 3 cells are selected.
Total of 4 cells are selected.
Total of 5 cells are selected.
```

(continues on next page)

(continued from previous page)

```
Total of 6 cells are selected.
Total of 7 cells are selected.
Total of 8 cells are selected.
Total of 9 cells are selected.
```

Write another loop that will print how many cells are selected in each row: (3 pts)

Listing 4: Sample Output

```
Total of 7 cells are selected.
Row 0 has 0 cells selected.
Row 1 has 0 cells selected.
Row 2 has 2 cells selected.
Row 3 has 2 cells selected.
Row 4 has 1 cells selected.
Row 5 has 2 cells selected.
Row 6 has 0 cells selected.
Row 7 has 0 cells selected.
Row 8 has 0 cells selected.
Row 9 has 0 cells selected.

Total of 8 cells are selected.
Row 0 has 0 cells selected.
Row 1 has 0 cells selected.
Row 2 has 3 cells selected.
Row 3 has 2 cells selected.
Row 4 has 1 cells selected.
Row 5 has 2 cells selected.
Row 6 has 0 cells selected.
Row 7 has 0 cells selected.
Row 8 has 0 cells selected.
Row 9 has 0 cells selected.
```

Update the code so that it prints the count in both rows and columns: (1 pt)

Listing 5: Sample Output

```
Total of 5 cells are selected.
Row 0 has 1 cells selected.
Row 1 has 1 cells selected.
Row 2 has 1 cells selected.
Row 3 has 1 cells selected.
Row 4 has 1 cells selected.
Row 5 has 0 cells selected.
Row 6 has 0 cells selected.
Row 7 has 0 cells selected.
Row 8 has 0 cells selected.
Row 9 has 0 cells selected.

Column 0 has 5 cells selected.
Column 1 has 0 cells selected.
Column 2 has 0 cells selected.
Column 3 has 0 cells selected.
Column 4 has 0 cells selected.
Column 5 has 0 cells selected.
Column 6 has 0 cells selected.
Column 7 has 0 cells selected.
Column 8 has 0 cells selected.
```

(continues on next page)

(continued from previous page)

Column 9 has 0 cells selected.

Update the code so that the program will also print how many cells are continuously selected in a row, if that number is greater than 2. (4 pts)

To do this, create a new variable (like `continuous_count`) that is reset to zero for each row. If the current grid location is one, then add one to `continuous_count`. If it is zero, check if `continuous_count` is greater than 2. If so, print the count. Regardless, reset it back to zero.

You'll also need to do the same `if` after the loop with the row is done. If the row goes up to the grid-edge, then the blocks-in-a-row won't trigger without this second check.

Listing 6: Sample Output

```
Total of 18 cells are selected.  
Row 0 has 0 cells selected.  
There are 7 continuous blocks selected on row 1.  
Row 1 has 7 cells selected.  
Row 2 has 0 cells selected.  
There are 4 continuous blocks selected on row 3.  
Row 3 has 4 cells selected.  
Row 4 has 0 cells selected.  
There are 4 continuous blocks selected on row 5.  
Row 5 has 4 cells selected.  
There are 3 continuous blocks selected on row 6.  
Row 6 has 3 cells selected.  
Row 7 has 0 cells selected.  
Row 8 has 0 cells selected.  
Row 9 has 0 cells selected.  
Column 0 has 1 cells selected.  
Column 1 has 3 cells selected.  
Column 2 has 3 cells selected.  
Column 3 has 3 cells selected.  
Column 4 has 3 cells selected.  
Column 5 has 1 cells selected.  
Column 6 has 1 cells selected.  
Column 7 has 1 cells selected.  
Column 8 has 1 cells selected.  
Column 9 has 1 cells selected.
```

2.12 Lab 12: Final Lab

This is it! This is your chance to use your creativity and really show off what you can create in your own game. More than just passing a test, in this class you actually get to do something, and create something real.

There are two options for the final lab. A “video game option” and a “text adventure option.”

Both options are divided into three parts. Each part should take about one week. The goals of this lab:

- Get used to working on longer programs by breaking them into different functions, classes, and files.
- Learn to search for code that does what you want, and adapt it into your program.
- Learn to break down large programs into smaller, easier to solve parts.
- Start creating a portfolio of work that you've done.
- See that programming can be a fun, creative outlet.

- Apply the concepts we learned, such as variables, expressions, if statements, loops, lists, arrays, and more.

This lab is worth 60 points. Each week, turn in what you have for the lab so you can “lock in” your score and get an idea of how you are progressing. When you get more done, resubmit the assignment.

You can wait until the end to turn in your work until the end of the semester, but that is a very risky plan.

2.12.1 Video Game Option

The video game option is the most popular option. Here are some videos of what students have done in prior years:

Using Images or Sounds from Other Sources

It is ok to use images and sounds from other sources if you cite the source.

Cite in the code comments what images and sounds you’ve used. So for every image or sound loaded, put a # tag in the line above with a citation on where it came from. I’ll expect more than just a URL, give me a name of the website as well.

For a professionally published game this would NOT be enough. You’d need to make sure you have license to use the work.

If you create your own image or sound, state that in the comments so I can be properly impressed.

Using Code from Other Sources

If you find code to use that doesn’t come from this website or from arcade.academy, cite it. Otherwise you’ll be plagiarizing, and flunked from the class.

Almost every semester I catch someone doing this. Please, just don’t.

Requirements for Part 1

- Cite in the code comments what images and sounds you’ve used. So for every image or sound loaded, put a # tag in the line above with a citation on where it came from. I’ll expect more than just a URL, give me a name of the website as well.
- If you find code to use that doesn’t come from this website or from arcade.academy, cite it. Otherwise you’ll be plagiarizing, and flunked from the class.

Each of the three parts for the final lab raises the bar on what your game needs to be able to do.

- Open up a screen.
- Set up the items to be drawn on the screen. Figure out the images that you want to use.
- Provide some sort of player movement or interaction via mouse, keyboard, or game controller. Get items on the screen to move, if applicable.
- Have something that is at least kind-of playable.

Tips:

- If your program will involve things running into each other, start by using sprites. Do not start by using drawing commands, and expect to add in sprites later. It won’t work and you’ll need start over from scratch. This will be sad.
- If you are coding a program like mine sweeper or connect four, do *not* use sprites. Since collision detection is not needed, there is no need to mess with sprites.

- While you can start with and use any of the example game code from this website or [arcade.academy](#), don't just turn these in as Part 1. You'll need to add a lot before it qualifies.
- A very common mistake is to make a call to `self.all_sprites_list.update()` and `self.physics_engine.update()`. This will cause you to warp through walls. You want to specifically update lists that don't include the player, like `self.enemy_list.update()`.
- Remember that any extra white-space around your image will throw off collision detection. Best to check and trim white space from images you use.
- [OpenGameArt.org](#) has a lot of images and sounds you can use royalty-free.
- [Kenney.nl](#) has many images and sounds as well.

Requirements for Part 2

For Final Lab Part 2, your game should be functional. A person should be able to sit down and play the game for a few minutes and have it feel like a real game. Here are some things you might want to add:

- Be able to collide with objects.
- Players can lose the game if something bad happens.
- On-screen score.
- Sound effects.
- Movement of other characters in the screen.
- The ability to click on mines or empty spots.

Requirements for Part 3

For the final part, add in the last polish for your game. Here are some things you might want to add:

- Title and instruction screens
- Multiple levels
- Lots of Sounds
- Multiple “lives”
- More types of enemies
- Power-ups
- Heat seeking missiles
- Hidden doors
- A “sweep” action in a minesweeper game or the ability to place “flags”

2.12.2 Text Adventure Option

Not interested in a video game? Continue your work from the “Adventure!” game.

Requirements for Part 1

- Rather than have each room be a list of [description, north, east, south, west] create a Room class. The class should have a constructor that takes in (description, north, east, south, west) and sets fields for the description and all of the directions. Get the program working with the new class. The program should be able to add rooms like:

```
room = Room("You are in the kitchen. There is a room to the east.", None, 1, None, None)
room_list.append(room)

room = Room("You are in the living room. There is a room to the west.", None, None, 0, None)
room_list.append(room)
```

Later the program should be able to refer to fields in the room:

```
current_room = room_list[current_room].north
```

- Perhaps expand the game so that a person can travel up and down. Also expand it so the person can travel northwest, southwest, northeast, and southeast.
- Add a list of items in your game.
 - Create a class called Item.
 - Add fields for the item's room number, a long description, and a short name. The short name should only be one word long. This way the user can type get key and the computer will know what object he/she is referring to. The description will be what is printed out. Like There is a rusty key here.
 - Create a list of items, much like you created your list of rooms.
 - If the item is in the user's room, print the item's description.
 - Test, and make sure this works.

Requirements for Part 2

- Change your command processing, so rather than just allowing the user to only type in directions, the user will now start having other options. For example, we want the user to also be able to type in commands such as get key, inventory or wave wand.
 - To do this, don't ask the user What direction do you want to go? Instead ask the user something like What is your command?
 - Split the user input. We need a variable that is equal to the first command they type, such as get and a different variable equal to the second word, such as key.
 - Use the split method that's built into Python strings. For example: command_words = user_command.split(" ") This will split what the user types into a list. Each item separated out based on spaces.
 - Update your code that processes the user typing in directions, to check command_words[0] instead of whatever you had before.
 - Add a get command.
 - Add a check for a get command in your if/elif chain that is now just processing directions.
 - Search the item list until you find an object that matches what the user is trying pick up.

3. If the object isn't found, or if the object isn't in the current room, print an error.
4. If the object is found and it is in the current room, then set the object's room number to -1.
3. Add a command for "inventory" that will print every object who's room number is equal to -1.
4. Add the ability to drop an object.
5. Add the ability to use the objects. For example "use key" or "swing sword" or "feed bear."

Requirements for Part 3

Expand the game some more. Try some of these ideas:

1. Create a file format that allows you to load the rooms and objects from a file rather than write code for it.
2. Have monsters with hit points.
3. Split the code up into multiple files for better organization.
4. Remove globals using a main function as shown at the end of the chapter about functions.
5. Have objects with limited use. Like a bow that only has so many arrows.
6. Have creatures with limited health, and weapons that cause random damage and have a random chance to hit.

2.12.3 Tips

- Commit and push your code often. It is not unusual for something bad to happen to the code while you are working on this assignment. Anything you commit and push we can recover. Don't work for more than a couple hours without doing this.
- I give a lot of in-class lab time for this project. Use it. Don't leave early.

2.13 Quantitative Reasoning Reflection

This is a reflection on quantitative reasoning. However, instead of turning it in via a PDF or MS Word document, we'll turn it in as a text document.

2.13.1 Directions

- Create a directory in your project called Quantitative Reasoning Reflection.
- Inside that directory, create a new file called `reflection.txt`. Don't create it as a Python file, because this isn't a Python program.
- In PyCharm, under the "File" menu, select "Settings...". Then find "Editor" and under that "Code Style." By default you could see a column guide set at 120 characters. Set this to 80 for now.
- Add your name, date, and a title to the document. Failure to do this is a three point penalty.
- Don't write text beyond the 80 character limit. (Occasionally hitting 85 isn't a big deal, but if you write 800 characters on a line, expect to get penalized.)
- Use paragraphs. Most paragraphs are around five sentences long. While this isn't a hard and fast rule, if your paragraph has 22 sentences, chances are, something's wrong.

- Answer each question below. Commit and turn in just like a regular program. Use complete sentences and proper spelling. Label each question. Put a blank line between each question.

2.13.2 Questions

Each question is worth five points. Make sure you have five good points as part of your answer. I'd suggest drafting out five points, and then fill in the points with proper sentences, grammar, and punctuation. Examples are encouraged.

1. Describe how you used multiple quantitative methods, techniques and algorithms to add functionality to your program. How did you decided what method to use? For example, how did you use lists? "If" statements? Loops? Classes? Functions? Sample pieces of code? How did you decide which methods to apply? Feel free to give examples.
2. Explain how you communicate your solution (from one programmer to another) when writing the code. Show how variable names matter, how function names matter, how comments and structure make your code easier to read.
3. How did you test and evaluate the accuracy of your code? When something didn't work, describe how you debugged what the error was. Also, how do you determine if data returned is correct data?
4. What are the limitations to using numerical methods to make decisions? For example, why can't we just replace humans with algorithms? Why can't we just evaluate students based solely on test scores? Why can't we evaluate teachers just on how their students do? What is the limitation to evaluating athletes just on their stats?