7-11-2018

Intro Programming Lecture Notes (Student Version)

Kyle Burke
Plymouth State University, paithanq@gmail.com

Lecture Notes: CS 2370: Intro Programming*

Lecture Notes - Student Version†

Kyle Burke

July 10, 2018

Abstract

Lecture notes for an introductory programming course in Python (version 3.x). There are many example problems suitable for "flipped" classes. This follows the order of Allen Downey's Think Python text. Some sections are skipped, but the basics are included through inheritance and polymorphism. No prior programming experience is expected.

Contents

*Kyle would always like to hear about how useful his notes are. If you have any comments about these, please email him at `paithanq@gmail.com` or leave a comment at `https://paithanq.blogspot.com/2018/07/intro-programming-lecture-notes.html`.

†Created with lectureNotes.sty, which is available at: `http://turing.plymouth.edu/~kgb1013/lectureNotesLatexStyle.php` (or, GitHub: `https://github.com/paithan/LaTeX-LectureNotes`).

0 Course Intro

0.1 The Course's Textbook

The text for this class is Dr. Allen Downey's "Think Python: How to Think like a Computer Scientist". The second edition of the book is available at `http://greenteapress.com/wp/think-python-2e/`[1].

These notes do not cover all parts of the textbook; I skip parts that I don't get to address during the semester, though I do cover some of the topics in more depth (or just provide additional examples and in-class exercises).

0.2 Acknowledgements

Incredible thanks to Allen Downey, my first CS professor back in the Fall of 1999. He made an impression in only one semester; his enthusiasm, attitude, and clarity are all qualities I have tried to emulate in my own teaching career. In addition, his introductory programming text (which these notes use) is excellent.

Thanks to Christin Wixson for helping me prepare these notes for adoption as an Open Educational Resource (OER). Her help has been invaluable in preparing this for the public, including navigating the ramifications of licenses. The entire Plymouth State repository of OERs is available at `https://digitalcommons.plymouth.edu/oer/`.

Most of all, thanks to the many students that took this course from me at Wittenberg University, Colby College, and Plymouth State. You have each taught me how to be a better teacher, and I wish I had all the tools I have now when I met you.

0.3 In Progress

- Translate the examples/questions from Python 2 to Python 3. The biggest recurring change here is that in Python 3, `print` is a function instead of just a special command.
- Adapt to the other changes in the second edition of Think Python.
- Add more Bonus Challenge problems throughout the notes so that there's (nearly) always something for students to be working on.
- Make sure the section numbers in here line up with the sections in Allen Downey's text.

⟨ Cover Syllabus and Expectations ⟩

1 The Way of the Program

Welcome to Introduction to Programming! In this class, you are going to learn to write programs! Exciting!

More goals:

[1] PDF (free and legal) available at `http://greenteapress.com/thinkpython2/thinkpython2.pdf`

- Learn an approach to problem solving, including:
  - stating problems clearly
  - creatively devising solutions
  - clearly expressing solutions
- Learn to think Algorithmically. "What's a good way to complete this task?"
- Learn to write computer programs. Sounds tricky, but we're going to do it!

1.1 Python

We are going to learn Python, a *high-level* programming language (like C++ or Java).

Def: *Low-Level Language* A *low-level language* is something written so the hardware can read it. Examples: machine code or assembly language.

Q: Why aren't we learning a low-level language?

A:

Q: How does a computer run a program?

A:

This slows down processing time a bit, but *greatly* speeds up the programming time.

Two flavors of high-level languages:

- Interpreted: Source Code $\overset{\text{Interpreter}}{\rightarrow}$ Output
- Compiled: Source Code $\overset{\text{Compiler}}{\rightarrow}$ Executable Program $\overset{\text{Executor}}{\rightarrow}$ Output TODO: draw picture

⟨ Say something about each! ⟩

Q: Which of the two is Python?

A:

Q: Why use a compiled language?

A:

Two modes for running Python code:

- Interactive Mode: "Chat" with Interpreter.

```
chevron
 >>> 2+2
4
>>>
```

- Script Mode: Store Programs in a `.py` file.

Q: When might you want to use each mode?

A:

1.2 What is a Program?

Q: What is a program?

A:

Common types of instructions:

- Input
- Output
- Math
- Conditional Execution
- Repetition

Every program you've ever seen is built from these pieces!

I will teach you to do this!

Since instructions need to be unambiguous, code must be precise and clean. Errors that humans could resolve, computers have a hard time with.

1.3 What is Debugging?

Debugging!

"Bugs": Rear Admiral Grace Hopper.

Def: *Debugging* Debugging: tracking down errors and fixing them.

Aside from Termintator-like scenarios, this is the biggest thing people complain about with computers.

Three types of errors:

- Syntax Error. Syntax are rules about structure. Example: `>>> 2 + ) 2`. Try it out!
  Like English grammatical errors.

Python checks first; one syntax error prevents WHOLE program from running.
You'll make lots of these in the beginning, but will get better.

- Runtime Errors. "Exceptions." Happen while program is running.
- Semantic Errors. Program runs and doesn't notice a problem, but it doesn't do what you wanted.
  Example: Want to print `Monkey!`, but type: `print('Donkey!')`.
  (Have students try this out.

Q: How do you know if you have errors in your code?

A:

Debugging is hard, but is a useful skill! Even outside of programming! Like forensics: need to find culprit. Error messages give you clues.

Debugging is an experimental science:

- Have an idea about what went wrong.
- Try to fix it.
- Run it again to see whether that worked.
- If not, maybe you need a new idea!

The book discusses debugging issues at the end of each chapter. Look there if you're having trouble!

Good idea: Purposefully make some mistakes! Learn what happens and become unafraid of errors! Let's do some of that now.

Q: Let's try causing some different types of exceptions. Give me a line of code that causes a TypeError.

A:
```
3 + "3"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError:  unsupported operand type(s)
for +:  'int' and 'str'
>>>
```

Q: Let's talk quickly about the info Python gave us back from this error. Why does it say "line 1"?

A: The error happened in the first line of the code. If we have a longer program in a script, this could be in a line other than 1.

Q: What is Python telling us in the last line of the error message?

A:

- TypeError: means it was trying to do something, but it couldn't because the types of the values weren't correct.
- Message (last line): It's saying that it doesn't know how to add an int to a str.

TODO: divide this up into things you expect they might know about versus things they certainly don't know about yet!

Q:

Try to generate the following errors in interactive mode. For each error type, see if you can come up with a line (or lines) of code that causes that type of error. (We've already seen `TypeError`, so I'm not including that.)

- `NameError`
- `ZeroDivisionError`
- `ValueError`
- `AttributeError`

Bonus: how do you cause a `KeyboardInterrupt`?

A:

Q: These are errors you might see later on in the course. Does anyone know how you might cause any of these?

- `ImportError`
- `IndexError`
- `KeyError`
- `FileNotFoundError`

A:

- `ImportError`:

```
>>> import bananaCheese
```

- `IndexError`:

```
>>> monkeys = ['tamarin']
>>> monkeys[1]
```

- `KeyError`:

```
>>> belugas = dict()
>>> belugas['cheese']
```

- `FileNotFoundError`:

```
>>> open('AshKetchum.txt')
```

[2]

Q: You can cause an *exception* that you create to occur. How does that work?

A: Do something like this:

```
>>> raise Exception("I don't know what
I'm doing!")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
Exception:  I don't know what I'm
doing!
>>>
```

[2] There are even more kinds of errors here: `https://www.tutorialspoint.com/python/standard_exceptions.htm`

[3]

1.4 Natural and Formal Languages

Def: *Natural Languages* Natural Languages are those people speak or use to converse with other people. These are designed by people, but come about naturally.

Def: *Formal Languages* Formal Languages are designed by people for a specific purpose, like programming languages.

Q:	What are other types of formal languages?
A:	

Formal languages are very strict about syntax.

[3] Source: `https://twitter.com/SashaLaundy/status/936661004137635840`

Q: What are examples of syntactically incorrect mathematical and chemical statements, but which use correct words and characters?

A:

Def: *Tokens* *Tokens* are basic elements of a language. For example: words, numbers, chemical elements

Def: *Structure* *Structure* is rules for arrangement of the language's Tokens.

Pulling apart statements to analyze structure and tokens is called *parsing*. We do this unconsciously in English!

Three properties of languages:

- Ambiguousness
- Literalness
- Redundant

Q: Which of these is:

- more ambiguous?
- more literal?
- more redundant?

A:

Informal → Formal: Poetry → Prose → Programs

Q: Reading Code is Hard! Why?

A:

You will learn to deal with all of these things! You will both READ and WRITE code well!

Debugging: you **will get frustrated**.

- People often respond to computers as though they were people!
- Errors are "Rude"! They will not always be clear!
- Computer is good at speed, not good at empathy!

Sleep on it if you have to! Take a walk. Change the music you're listening to. Switching your environment can often help you change the way you're thinking.

2 Variables, Expressions and Statements

2.1 Values and Types

Def: *Values* *Values* are basic data used by a program.

```
>>> 1 + 2
3
```

That uses the values 1 and 2. In lab, used value: `'Hello, World!'`

There are different value types:

- 1,2: integers
- `'Hello, World!'`: string. "string" of letters. Strings are always in quotes!

Sometimes you need double-quotes in strings


```
>>> 'That's all folks!'
Syntax Error!
>>> "That's all folks!"
```

Python can identify types for you!

```
>>> type('Hello, World!')
<class 'str'>
>>> type(42)
<class 'int'>
```

Q: What's going to happen here?

```
>>> type(3.25)
```

A:

`float` means Floating-Point. That means it's a number with a fractional part.

Q: What's going to happen here?

```
>>> type('3.25')
```

A:

Q: Why?

A:


```
>>> print(42)
42
>>> print('42')
42
>>> print(1,000,000)
1 0 0
```

Q: What happened there?

A:

Q: What kind of error did I get?

A:

```
>>> type(1,000,000)
Traceback ...
```

Q: What happened there?

A:

Q: What kind of error did I get?

A:

2.2 Variables

Def: *A* variable *is "named storage" for a value.*

Create variables with assignment statements:

```
>>> message = 'I love variables!'
>>> number = 42
>>> pi = 3.14...
```

⟨ Add and describe: state diagram! ⟩

Q: What will happen when I try:

```
>>> print(message)
```

?

A:

Q: What about:

```
type (pi)
```

?

A:

2.3 Variable Names and Keywords

Variable Names must be:

- Legal. Illegal things include:
  -
```
>>> 3age = 84
SyntaxError:  invalid syntax.
```
  Variable names must begin with a letter!

– `>>> phone@home = 9375555555`

`@` is an illegal character!

– `>>> important variable = 42`

No spaces! Use snake_case!

`>>> important_variable = 42`

– `>>> import = 'cupcakes'`

`import` is a keyword: full list in the book.

- Meaningful!

  Variable names should describe the purpose or role of the variables.

Q: Which of the following variable names is best?

- `number = 5`
- `number_of_tigers = 5`
- `number_of_tigers_at_zoo = 5`

A:

A common convention is to drop the "`ber_of`" part, so you could name your variable `num_tigers_at_zoo` and that's completely fine.

2.4 Statements

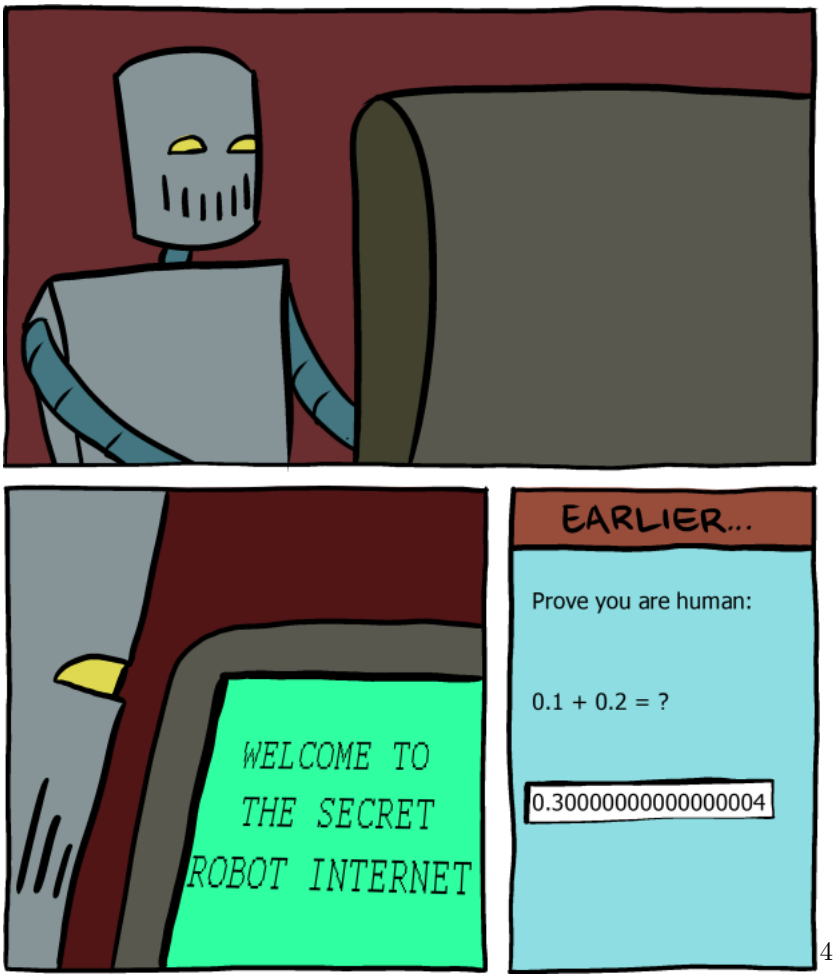
2.5 Operators and Operands

We can use operators with values and variables. Math operators include: `+`, `-`, `*`, `/`, `**`.


```
>>> 33-28
5
>>> 3**2
9
>>> number-3
39
>>> 1/2
0
>>> 1.0/2
.5
>>> 1.0 // 2
0.0
>>> .1 + .2
.30000000000000004
```

Q: What's going on with these last two?

A: The `//` operator divides and removes any remainders. The last one is the closest number Python can recognize near .3.



[4]. Used with permission.

2.6 Expressions

These are all expressions: combinations of values, variables and operators.

```
(2*7)*(1+2)
```

In interactive mode, after typing in an expression, interpreter evaluates and displays.

Script mode: no automatic displaying. Alone, an expression doesn't do anything.

[4]Source: ://www.smbc-comics.com/comic/2013-06-05

2.7 Order of Operations

Q: Without trying them out, what is the result of each of these? (Write them on the board.)

- `2**2*2`
- `(2**2)*2`
- `2**(2*2)`
- `5/2 + 1`
- `1*1+1**1`
- `1*(1+1)**1`

A:

I always use parameters to make things clear!

2.8 String Operators

Some of these work on strings too!

```
>>> last_name = 'Stickney'
>>> print('2 ' + last_name)
2 Stickney
>>> print(3*'Monkey')
MonkeyMonkeyMonkey
```

String "adding" is known as *concatenation*.

Sometimes you want to print strings and non-strings in the same line. `print` can do this!


```
(In script)
number_of_squirrels = 5
print ''I have'', number_of_squirrels,
''squirrels, isn't that nice?''
```

Q: What happens when you run this?

A:

Q: What if I change the 5 to a 75 and rerun it?

A:

2.9 Comments

Code can be tough to read!

- Expressions are complex
- Whole sections can be complex

Solution: Comments!

Def: *Comment* A *comment* is a note to yourself and other programmers, but not to Python.

In script:

```
#compute the circumference of the circle
circumference = 2 * radius * pi
```

Or it can be on the same line:

```
area = pi * radius**2 #area of a circle
```


symbol tells the interpreter to ignore the rest of the line. For humans, but not machines!

Q: What is wrong with these?

- `monkey = 16 #assign monkey to 16`
- `monkey = 16 #monkey is driving time to Indianapolis in hours`

A:

Q: What would be better?

A:

TRADEOFF! Variable names vs. comments

- longer variable names: easier to understand the role
- shorter names: expressions are easier to visually parse

In general, I err towards longer names[5].

3 Functions

Def: *Function* In programming, a *function* is kind of like a subprogram:

- Like a program, it also a sequence of instructions
- ...But, has a name like a variable

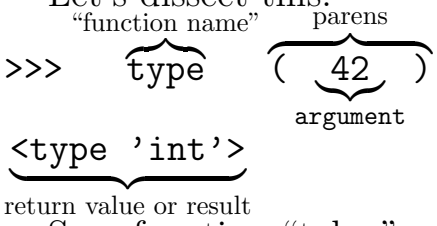
[5] Excellent list of useless comments here: `http://www.neobytesolutions.com/the-least-useful-comments-ever/`

3.1 Function Calls

You can use or *call* functions inside a program once they're defined.

```
>>> type(42)
<type 'int'>
```

Let's dissect this:



Say: function "takes" argument and "returns" the result.

3.2 Type Conversion Functions

Some other functions are already defined in Python. For example: Type Conversions! Sometimes you want to convert a variable from one type to another.

```
>>> int(3.0)
3
>>> int(2.7182817284)
2
>>> int(42.7)
42
```

Q: How would you describe what the `int` conversion function does to floating-point values?

A:

Q: **Test-Focus**: What could we do to test this in a *new* way?

A:


```
>>> int(-3.45)
-3
```

Q: What do you think it does now?

A:

Q: For each of these, guess at what you think the value is that's returned, then try it out:

```
>>> float(3)

>>> float(42.7)

>>> str(32)

>>> str(32.0)
```

A:

Q: Do the same for these:

```
>>> int("34")

>>> float("34.0")

>>> int("34.0")

>>> float("34")
```

A:

Q: What will happen with the call `int('monkey')`?

A:

3.3 Math Functions

Also, you might want to use some 'popular' math functions: logarithms, trig, square root. They exist in `math` module.

Def: *module* A *module* is a collection of pre-defined variables and functions.

Loading the module is easy!

```
>>> import math
```

Just like a variable, this is cleared when we restart interactive mode by running a script. If you use `math` in your script, put `import math` at the top.

```
>>> math.pi
3.1415926535897931
```

This is the closest number to pi that Python can represent, stored in the variable math.pi. Many useful math functions! Say we want to find the sine of 1.5 (radians).

```
>>> math.sin(1.5)
.997...
```

"dot notation"

`math` • `sin(1.5)`

package name — variable or function call

We can use variables as the arguments!

```
>>> radians = 1.5
>>> math.sin(radians)
.997...
```

Q: `math.sin` expects the argument to be in radians. What if I have the number of degrees instead?

A:

Help me fill in the following code!

```
(In script)
import math
degrees = 200
radians = ...
print(math.sin(radians))
```


This should print: `-.3420`

Q: How hard is it to change this script to try it with a different number of degrees?

A:

3.4 Composition

You can also use expressions inside arguments!

Q: In interactive mode, how do I compute this: $\sqrt{1+50}$?

A:

You can also use function calls inside other expressions!

Q: What about $\frac{\sqrt{3}}{6}$?

A:

If I wasn't sure that my solution was correct, I could check in two lines:

```
>>> x = math.sqrt(3)
>>> x/6
...
```


Q: What about: $\sin\left(\frac{1.5^2}{2}\right) - 1$?

A:

```
>>> x = 1.5**2
>>> y = x / 2
>>> z = math.sin(y)
>>> z - 1
-.997...
```

Q: What about: $\cos\left(\sqrt{1.21}\right)$?

A:

Composition! You can have multiple levels of composition!

Q: We saw the issue with trying to convert `"34.0"` into an `int`. However, `int(34.0)` isn't a problem. How could we use this to get around the first problem?

A:

3.5 Adding new Functions

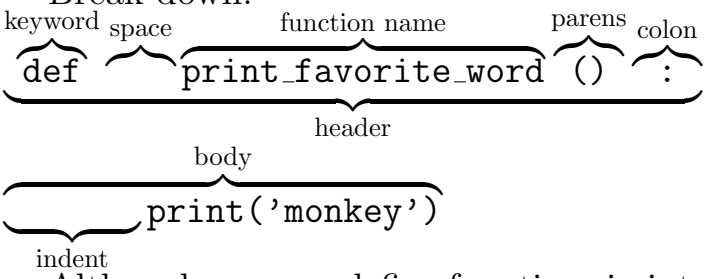
Q: What if no function does what you want?

A:


```
(In script!)
def print_favorite_word():
    print(''monkey'')
```

Let's look more closely!

Break down:



Although you can define functions in interactive mode, it's better to use script mode!

```
>>> print_favorite_word()
'monkey'
```

A function body can have multiple lines!

```
(In script!)
def print_favorite_words():
    print('monkey')
    print('bonus')
    print('awesome')
```

Run the script, then:

```
>>> print_favorite_words()
monkey
bonus
awesome
```

What happens if you define `print_favorite_word` twice?

```
(Add to bottom of script)
def print_favorite_word():
    print('awesome')
```


Q: Which word will be printed when I call `print_favorite_word`?

A:

⟨ Delete the second definition from the script. ⟩

Q: How could you write the function, `print_kyles_favorite_word_twice` that printed out my favorite word twice?

A:

```
>>> print_kyles_favorite_word_twice()
monkey
monkey
```

Q: When I wake up tomorrow, what if I change what my favorite word is? I change `print_favorite_word` but not the others! Can I change `print_kyles_favorite_word_twice` so that this isn't a problem?

A:

```
>>> print_kyles_favorite_word_twice()
monkey
monkey
```

Another form of function composition!

3.6 Definitions and Uses

(I skip this section in class.)

3.7 Flow of Execution

Let's look at how this works! "Flow or Thread of Execution!"

⟨ Demonstrate flow, first on `print_favorite_word`, then from `print_kyles_favorite_words`. ⟩

Function calls are like detours. When you are reading a program, be sure to follow the flow. At a function call, read through the call before continuing.

3.8 Parameters and Arguments

So far, we haven't shown how to use arguments!

```
              argument
math.sqrt(  25  )
```

⟨ Load the chapter3 code! ⟩

For exmaple, maybe I want a function that greets a friend:

```
(Interactive)
>>> greet('Bob')
Hi, Bob!
>>> greet('LuAnn')
Hi, LuAnn!
```

Solution: Parameters! (We'll work on defining `greet` in a little bit; hold on.)

Def: *Parameters* *Parameters* are variables that are assigned to the arguments of a function.

```
(On board)
def print_twice(message):
    print(message)
    print(message)
```

Q: What will happen when we do this?

```
>>> print_twice('spam')
```

A:

Try it out! Write the function in script mode and then call it in interactive mode!

Q: What does the following function call do?

```
>>> print_twice('Spam' * 4)
```

A:

Q: Follow Up!
What is the value of `message` inside the function?

A:

Q: What about the following?

```
my_name = 'Kyle'
greet(my_name)
```

A:

3.9 Variable Locality

Note: parameters are *local*! They do not persist outside of the function!


```
>>> message
Error!
```

A common error with parameters is to redefine the variable!

```
def print_twice(message):
    message = 'spam'
    print(message)
    print(message)
```

Now `print_twice` only works with `'spam'`!

```
>>> print_twice('spam')
spam
>>> print_twice('spamalot')
spam
```

This is common! Don't redefine your parameters!

```
>>> greet('Bananaman')
Hi, Bananaman!
>>> brag_about('Elman Bashar')
Wow, Elman Bashar is definitely the coolest
person I know.  They are just so awesome!
>>>
```

Q: How can we write the `greet` function?
Header: `def greet(name):`
Bonus Challenge: `brag_about`

A:

Q: Problem here?

A:

Q: How can we fix this?
Hint: use string concatenation.

A:

Q: What if `name` is not a string?

```
>>> greet(5)
Error!
```

Q: How can we fix this?
Hint: use type conversion!

A:

Try out the following cases

```
>>> name = 'Kyle'
>>> greet(name)
Hi, Kyle!
>>> greet('name')
Hi, name!
>>> greet(Name)
Error!
```



```
>>> greet_whole_name('Kyle', 'Burke')
Hi, Kyle Burke!
>>> greet_whole_name('Ash', 'Ketchum')
Hi, Ash Ketchum!
>>> super_brag_about('Elman', 'Bashar')
Wow, Elman Bashar is definitely the coolest
person I know.  They are just so awesome!  Elman
can jump through hoops of fire and once bested a
Siberian Horsetiger in a game of Chessboxing!
```

Q: How can I write the body of the `greet_whole_name`? Header: `def greetWholeName(first_name, last_name):` Bonus Challenge: `super_brag_about`.

A:

Or, could create a new variable inside the body.

```
def greet_whole_name(first_name, last_name):
    whole_name = str(first_name) + " " +
str(last_name)
    print('Hi, ' + whole_name + '!')
```

Q: Did I make a mistake by not putting `str(whole_name)` in that last line?

A:

Q: Which one do I like better?

A:

Q: Can we do even better?

A:

Q: Which function body does the last line of the last `greet_whole_name` look like?

A:

Q: Could we replace with a call to `greet`?

A:

Q: Why is this considered better?

A:

Actually a big part of Software Design. When I change code, try to change it in fewest places possible. Good code allows for this!


```
>>> first_name = 'MC'
>>> greet_whole_name(first_name, 'Frontalot')
Hi, MC Frontalot!
>>> print(first_name)
MC
>>> print(whole_name)
Error!
```

Q: Why did I get that error?

A:

Q: Why not with `first_name`?

A:

```
>>> greet_whole_name('Mr.', first_name)
Hi, Mr.  MC!
```

3.10 Stack Diagrams

Things can get confusing! Sometimes we need to keep track of variable locality. Use a *stack diagram!*

⟨ Do stack diagram for the last code example. `__main__` ⟩

⟨ One more for `greet('Steve')`. ⟩

Your turn!

```
(script)
def concatenate_twice(message_one, message_two):
    full_message = message_one + message_two
    print_twice(full_message)
```


Q: What do the stack diagrams look like for this call?

```
>>> spanish = 'Hasta la vista, '
>>> english = 'baby.'
>>> concatenate_twice(spanish, english)
```

Try it out and check with your neighbors!

Stack diagrams very helpful for figuring out what went wrong! Use them on projects!

```
(interactive mode)
>>> print_product(5, 3)
5 times 3 = 15
>>> print_product(9, -7)
9 times -7 = -63
>>> print_power(3, 3)
3 raised to 3 = 27
>>> print_power(-1, 7)
-1 raised to 7 = -1
```

Q: Implement `print_product`! Bonus challenge: `print_power`.

A:


```
(interactive mode)
>>> print_energy(50)
50 grams contains 4493775893684088200 available
joules.  >>> print_energy(.001)
.001 grams contains 89875517873691.77 available
joules.
>>> print_quadratic_roots(4, 0, -4)
4x^2 + 0x + -4 has roots 1.0 and -1.0.
>>> print_quadratic_roots(1, 3, 2)
1x^2 + 4x + 2 has roots -0.5857864376269049 and
-3.414213562373095
>>> print_quadratic_roots(1, 2, 3)
Error!
```

Q: Implement `print_energy`! Hint: $c = 299,792,458 m/s$.
Bonus challenge: `print_quadratic_roots`.

3.11 Fruitful and Void Functions

There's still something different about the `math` functions we used!

```
>>> math.sqrt(16)
4.0
>>> math.sqrt(math.sqrt(16))
2.0
```

Q: What's different?

A:

Def: *Fruitful and Void* *Fruitful* functions return a value. *Void* functions do not (they actually return `None`).

I could write that last part as:

```
>>> x = math.sqrt(16)
>>> math.sqrt(x)
2
```

I wanna do that too! So far: we can't! Try:

```
>>> greeting = greet('Hercule')
Hi, Hercule!
>>> print(greeting)
None
```

`None` is a special value: represents no actual, usable value (nothing was assigned). It even has it's own type!

```
>>> type(None)
<type 'NoneType'>
```

Def: *void* The functions we've written are *void*, which means they have no return value.

We will write fruitful functions soon... in chapter 5.

3.12 Why Functions?

Q: Why do we want to write functions anyways?

A:

4 Repetition and Turtles

In programming, there is lots of repetition!

For example, we might want a function that greets multiple times!

```
>>> greet_n_times('Steve', 5)
Hi, Steve!
Hi, Steve!
Hi, Steve!
Hi, Steve!
Hi, Steve!
>>> greet_n_times('Michaelangelo', 3)
Hi, Michaelangelo!
Hi, Michaelangelo!
Hi, Michaelangelo!
```

4.1 TurtleWorld

In the next lab, you will steer a turtle! Turtles like to draw! You will need to copy and paste the following lines from the syllabus:

```
(In script)
import sys
sys.path.append('Q:\\Computer
Science\\PythonPackages\\swampy.1.1')
from TurtleWorld import *
```

WARNING: do not name your script 'TurtleWorld.py'

Now I can create a Turtle and get it to do stuff! Notice: some PascalCase coming up—used for class names. Classes are more complicated types.

```
>>> world = TurtleWorld()
>>> bob = Turtle()
>>> fd(bob, 100)
>>> rt(bob)
>>> fd(bob, 100)
```


Q: What would it take to draw a square with side length 50 and end up facing the same way we started?

A:

4.2 Simple Repetition

Lots of repetition! Same thing four times! Let's simplify with `for`!

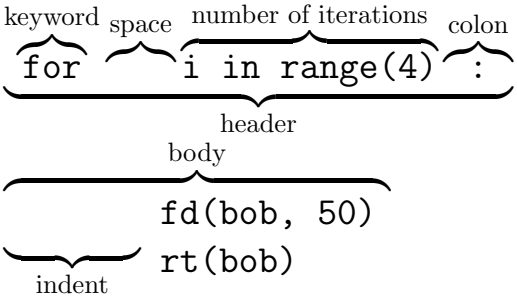
```
>>> for i in range(3):
        print('Monkey')

Monkey
Monkey
Monkey
```

Q: How can we use this to condense the square-drawing program?

A:

Break down:



This is known as a *for loop*.

Q: Why is this helpful?

A:

Q: Change it, how?

A:

Q: How many places would we have to change it before? How many now?

A:

Huge improvement!
We can put a loop inside a function!

Q: How would you write `greet_n_times`? Header:

```
def greet_n_times(name, n):
```

A:

Let's do the same for our turtles! Let's draw squares with side length 100.

4.3 Exercises

4.4 Encapsulation

```
>>> bob_draw_square()
<bob draws a square of side length 100>
>>> bob_draw_plus_sign()
<bob draws a plus sign>
```

Q: Implement it! Bonus challenge: `bob_draw_plus_sign`

A:

```
>>> bob_draw_hexagon()
<bob draws a hexagon with side length 100>
>>> bob_draw_star()
<bob draws a star>
```


Q: Implement `bob_draw_hexagon`! Hint: try: `rt(bob, 30)`. Bonus challenge: `bob_draw_star`

A:

Next: let's make these functions more useful!

We might do a lot of square drawing. Perhaps we are drawing lines for four-square! Before our function creation, our code might have looked like:

```
  for i in range(4):
    fd(bob, 100)
    rt(bob)
.
.
.
for i in range(4):
    fd(bob, 100)
    rt(bob)
.
.
.
for i in range(4):
    fd(bob, 100)
    rt(bob)
```

We simplified this by replacing the repetitive code with function calls! Now our code might look like:

```
  bob_draw_square()
.
.
.
```



```
bob_draw_square()
.
.
.
bob_draw_square()
```

Known as *Encapsulation*.

Def: *Encapsulation* *Encapsulation*: repeated code block → put code in function, replace blocks with function call.

4.5 Generalization

Now, what if we have two turtles? Could have two functions.

```
>>> bill = Turtle()
>>> fd(bill, 150)
>>> bob_draw_square()
>>> bill_draw_square()
```

Q: Is there a better option?

A:

```
>>> draw_square(bob)
>>> draw_square(bill)
```


Q: Implement it! Header:

```
def draw_square(turtle):
```

Bonus Challenge: update to `draw_hexagon`, `draw_star`, and `draw_plus_sign`

A:

This is *Generalization*!

Def: *Generalization* *Generalization*: function specific to one value → function applicable to different values by adding parameters

Q: Can we generalize `draw_square` further?

A:

```
>>> draw_square(bill, 25)
>>> draw_hexagon(bill, 35)
>>> draw_plus_sign(janelle, 100)
>>> draw_star(bob, 50)
```

Q: Implement it! Header: `def draw_square(turtle, side_length):` *Bonus Challenge*: same for `draw_hexagon`, `draw_plus_sign`, `draw_star`.

Q: Which two functions are very similar?

A:

Q: Should we generalize these into one function?

A:

```
>>> draw_polygon(bob, 9, 30)
>>> fd(bob, 100)
>>> draw_polystar(bob, 7, 30)
```

Q: Which parameter are we adding?

A:

Q: Implement it! Header:

```
def draw_polygon(turtle, num_sides,
side_length):
```

Hint: total angle degree is 360.
Bonus Challenge: `draw_polystar`

A:

```
>>> draw_box_stack(bob, 5, 30)
>>> rt(bob)
>>> fd(bob, 100)
>>> draw_box_grid(bob, 5, 30)
```

4.6 Interface Design

Let's try drawing a circle! Let's approximate (often the best we can do with computers) by drawing a polygon with lots of short sides.

```
>>> for i in range(180):
    fd(bob, 4)
    rt(bob, 2)
```

Q: Is this something we might do a lot?

A:

Q: What should we do, then?

A:

```
>>> bobDrawCircle()
```

Q: Do it! Header:

```
def bobDrawCircle():
```

A:

Q: Can we rewrite this by calling `drawPolygon`?
Hint: in the book

A:

Q: What should we do to improve `bobDrawCircle`?

A:

First, let's rewrite to use any turtle.


```
>>> drawCircle(bob)
>>> sally = Turtle()
>>> drawCircle(sally)
```

Q: Implement it! (Change the original one!) Header:

```
def drawCircle(turtle):
```

A:

Now the radius! More tricky, need to calculate the circumference!

```
>>> drawCircle(bob, 50)
>>> drawCircle(sally, 125)
```

Q: Implement it! (Change the original one!) Header:

```
def drawCircle(turtle, radius):
```

Hint: last line is: `drawPolygon(turtle, numberOfSides, sideLength)`
Double Hint: It's in the book!

A:

Q: Should we generalize to add `numberOfSides` as a parameter?

Q: Is this appropriate?

A:

```
>>> drawCircleBad(bob, 75, 50)
>>> drawCircleBad(bob, 100, 12)
```

Q: Is this appropriate?

A:

`turtle` and `radius` describe **what** is to be drawn.
`numberOfSides` describes **how**
functions should always work. Arguments should tell them what to do, but not necessarily how to do it. Assume they know best how to do it already!

Def: *Interface* A function's *interface* includes what a function does, arguments and a return value, if any.

4.7 Refactoring

4.8 Docstrings

(Order swapped with Develpment Plan.)

Interfaces should be as uncluttered as possible. Use *docstring* to describe a function's interface!

Def: *docstring* A *docstring* is a string used to describe a python function's interface. It should be in triple quotes and belongs directly after the header.


```
(On board)
def greet(name):
    '''Prints out a greeting to someone called
name.'''
    print('Hi, ' + str(name) + '!')
```

Notice, it's still indented over!

Q: Try adding a docstring to `drawSquare`!

A:

docstrings:

- use triple quotes, so it can be a multi-line string!
- explains *what*, but **not** *how*.
- shouldn't be too hard to write. If it is, indication of inappropriate function.
- do for all functions! I will now deduct points if they're not there!

An Interface is a contract between the function and the user.

- User *promises* to provide appropriate arguments.
- Function *promises* what it will do/accomplish.

4.9 Development Plan

Now we have a good plan for writing programs:

- Write a small program with no functions.
- Get it working.
- Encapsulate it into a function.

- Generalize the function by adding parameters.
- Start writing the next part of your program outside of any function.
- Repeat the steps above to build a complete set of functions.
- Look for places to rework your functions to improve interfaces and remove code re-use.

When writing functions, use the following order:

- Write the header.
- Write the docstring.
- Add the body.

This way you know what you're expecting your function to do logistically.

5 Conditionals and Recursion

5.0 The Modulus Operator

New Math:

```
>>> 13 / 4
3 ("Quotient")
>>> 13 % 4
1 ("Remainder")
```

Def: % is the modulus operator.

```
>>> 12 % 3
0
>>> 100 % 20
0
```

Q: When is $x\%y$ zero?

A:

5.1 Boolean Expressions

New Data Type: Booleans! Used to represent "Truthiness". Only two values: `True` and `False`

```
>>> True
True
>>> False
False
>>> type(True)
<type 'bool'>
>>> not False
True
>>> x = True
>>> x
True
>>> not x
False
```

We can have expressions that return a boolean value! For example, test whether two things are equal!

```
>>> 5 == 5
True
>>> 5 == 6
False
>>> 5 < 6
True
>>> 5 > 6
False
>>> 5 >= 6
False
>>> not (5 == 6)
True
```

Note: *HAVE* to use double equals when testing for equality!

```
>>> 5 = 6
Error!
```



```
>>> x = 4
>>> y = 8
>>> not (x == y)
True
>>> x != y
True
```

5.2 Logical Operators

We also have operators that take booleans: `and`, `or`, `not`.

```
>>> True or False
True
>>> False or False
False
>>> True and False
False
>>> True and True
True
>>> not (False and False)
True
```

Q: What are some other expressions equivalent to: `(x == 10) or (x > 10)`?

A:

Q: Expression equivalent to $x \in [0, 10]$?

A:

Q: If I have two ints, `a, b`, how do I test whether `a` is divisible by `b`?

A:

Q: Test whether int `x` is odd?

A:

```
>>> print_about_positivity(5)
It is True that 5 is positive.
>>> print_about_positivity(-22.3)
It is False that -22.3 is positive.
>>> print_about_positivity(0)
It is False that 0 is positive.
>>> print_all_about_sign(42)
It is True that 42 is positive.
It is False that 42 is negative.
It is False that 42 is zero.
>>>
```

Q: Write `print_about_positivity(number)`! Bonus challenge: `print_all_about_sign`

A:


```
>>> print_whether_in_closed_interval(5, 3, 1200)
It is True that 5 is in the interval [3, 1200].
>>> print_whether_in_closed_interval(-5, -3, 1200)
It is False that -5 is in the interval [-3,
1200].
>>> min(10, 100)
10
>>> print_whether_in_closed_interval(23, 30, 4)
It is True that 23 is in the interval [4, 30].
>>>
```

Q: Get `print_whether_in_closed_interval` to work as in the first two examples. Header: `print_whether_in_closed_interval(x, low, high)`. Bonus challenge: get it to work for the harder case, where the lower interval boundary might be last.

A:

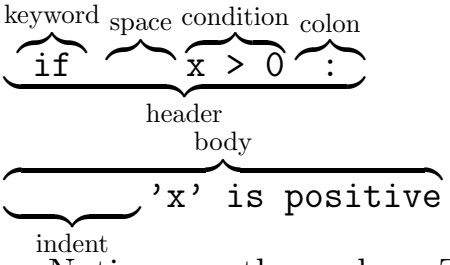
5.3 Conditional Execution

Often, we want to execute a block of code based on the value of a boolean expression. For example, might want to print out whether the value in a variable is positive.

```
(On board)
if x > 0:
    print('x is positive.')
```

This is known as a Conditional! Let's break it down!

Break down:



Notice: another colon. This means we have a compound statement!

Def: *Compound Statement* A *compound statement* is a statement that contains a header and a body consisting of other substatements.

There must be at least one statement inside the body of a compound statement!

As an example, let's modify `draw_polygon` by adding a conditional (and a `docstring` and comments).

```
(On board)
def draw_polygon(turtle, num_sides, side):
    '''Specified turtle draws a regular polygon
with num_sides sides, each of length side.'''
    if (num_sides) > 1:
        #draw the polygon!
        for i in range(num_sides):
            fd(turtle, side)
            rt(turtle, 360.0/num_sides)
    if (num_sides) <= 1:
        #print a message about not doing it
```

Q: Will this run?

A:

Q: What if we don't know what we want to write yet?

A:

`pass` is a line of code that does nothing.

```
>>> world = TurtleWorld()
>>> raphael = Turtle()
>>> draw_polygon(raphael, 5, 50)
>>> rt(raphael)
>>> pu(raphael)
>>> fd(raphael, 100)
>>> pd(raphael)
>>> draw_polystar(raphael, 7, 50)
>>> pu(raphael)
>>> fd(raphael, 100)
>>> pd(raphael)
>>> draw_polystar(raphael, 8, 50)
>>>
```

```
>>> better_print_whether_in_closed_interval(5, 0,
10)
Yes, it is!
>>> x = 13
>>> better_print_whether_in_closed_interval(x, -3,
3)
No, it isn't!
>>> better_print_whether_in_closed_interval(10, 0,
10)
Yes, it is!
```


Q: Implement it! Header:

```
def better_print_whether_in_closed_interval(number,
lower, upper)
```

Challenge: print out better message. Example: `Yes, 5 is between 0 and 10!`

5.4 Alternative Execution

We are often doing one thing or something else. Trick: use `else`.

```
(On Board)
if x > 0:
    print('x is positive.')
else:
    print('x is either negative or zero.')
```

Q: Replace our implementation of `better_print_whether_in_closed_interval` so that it uses `else`.

```
>>> better_print_whether_in_closed_interval(6, 10,
0)
Yes, it is!
>>> print_whether_monkey("sneasel")
sneasel isn't a monkey!
>>> print_whether_monkey("monkey")
monkey!
>>>
```

Q: Change yours so the same things happens! Also, code up `print_whether_monkey`

5.5 Nested Conditionals

(order swapped with Chained Conditionals)

We can *nest* conditionals!

```
(On Board)
if x > 0:
    print('x is positive.')
else:
    if x < 0:
        print('x is negative.')
    else:
        print('x is zero')
```

Q: Does the following code work?

```
(On Board)
if x > 0:
    print('x is positive.')
if x < 0:
    print('x is negative.')
else:
    print('x is zero.')
```

Q: Why not?

A:

5.6 Chained Conditionals

Alternatively, we can use `elif`. Called a *chained conditional*.


```
(On Board)
if x > 0:
    print('x is positive.')
elif x < 0:
    print('x is negative.')
else:
    print('x is zero.')
```

There can be an unlimited number of `elif`s in one conditional!

```
(On Board)
if x > 0:
    print('x is positive.')
elif x < 0:
    print('x is negative.')
elif x == 0:
    print('x is zero.')
else:
    print('x must not be a number!')
```

Q: What happens if I start with an `elif` instead of an `if`?

A:


```
>>> print_about_sign(13)
13 is positive.
>>> print_about_sign(-2210)
-2210 is negative.
>>> print_about_sign(5-5)
0 is zero.
>>> print_about_type(35)
35 is an integer.
>>> print_about_type('gibbon')
gibbon is a string.
>>> print_about_type([1, 2, 3])
[1, 2, 3] is a list.
```

Q: Write `print_about_sign(number)`! You may use either nested or chained conditionals! Bonus Challenge: `print_about_type`.

A:

Here's some *very* contrived examples that tends towards Nested Conditionals:

```
>>> print_about_integer(10)
10 is positive and not a perfect square.
>>> print_about_integer(16)
16 is positive and a perfect square.
>>> print_about_integer(-4)
-4 is negative and even.
>>> print_about_integer(-27)
-27 is negative and odd.
>>> print_about_integer_or_string('hi')
hi comes before "kyle" alphabetically.
>>> print_about_integer_or_string(35)
35 is odd.
>>> print_about_integer_or_string('zilwaukee')
zilwaukee comes after "kyle" alphabetially.
```


Q: Implement `print_about_integer`. Bonus challenge: `print_about_integer_or_string`. Hint: you can use less-than and greater-than to compare strings!

A:

One more example. I did this one with a chained contitional:


```
>>> print_about_turtle('Raphael')
Raphael is rude.
>>> print_about_turtle('Donatello')
Donatello does machines.
>>> print_about_turtle('Leonardo')
Leonardo leads.
>>> print_about_turtle('Michaelangelo')
Michaelangelo is a party dude.
>>> print_about_turtle('Shredder')
Shredder is not a teenage mutant ninja turtle.
>>>
```

Q: Implement `print_about_turtle`! Bonus challenge: TODO

A:

5.7 Recursion

We have functions call other functions. We can also have functions call themselves! Whoa!!!!!

```
(On Board)
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

Q: What's going to happen when I make this function call: `countdown(3)`?

```
>>> countdown(3)
3
2
1
Blastoff!
```

5.8 Recursive Stack Diagrams

A stack diagram will help figure out what happened!

⟨ Draw out the stack diagram for this! ⟩

Def: *Recursion* *Recursion* is the process of a function calling itself.

I could write this differently

```
(On Board)
def countdown_alternative(n):
    if n <= 0:
        print('Blastoff!')
        return
    print(n)
    countdown_alternative(n-1)
```

```
>>> countdown_alternative(3)
3
2
1
Blastoff!
```

Q: What does `return` do?

A:

5.9 Infinite Recursion

```
(On Board)
def keep_going():
    print(''Let's keep going!'')
    keep_going()
```

Q: What happens when I call `keep_going()`?

A:

Q: Why?

A:

Def: *Infinite Recursion* *Infinite Recursion* is a sequence of recursive calls that will never terminate.

Q: What if we swap the two lines in the body of `keep_going`?

```
>>> keep_going_backwards()
```

A:

You can press `Ctrl + C` to stop execution. Demonstrate!

Break down parts of `countdown`:

```
def countdown(n):
    if n <= 0:                  } base case
        print('Blastoff!')      }
    else:                       }
        print(n)                } recursive case
        countdown(n-1)          }
```


[6]

When you are writing recursive functions, make sure you have both base and recursive cases!

Recursive Function Writing Plan: (with `countdown` as the example)

1. Write the condition for the base case. (`n <= 0`)
2. Write the body of the base case (`print('Blastoff!')`).
3. Write the recursive *call*.
4. **Leap of Faith**: What should that recursive call do? Assume it does that!
5. Little bit of work: Write the code that uses that recursive call to solve the problem.

[6] The earliest version of this I found was at `http://www.csd.uwo.ca/courses/CS2120a/class15.html`, but I first saw it at `https://www.reddit.com/r/ProgrammerHumor/comments/5geas6/snack_overflow/`.

Q: What is the leap of faith for `countdown`?

A:

Q: In `countdown`, what was the little bit of work that is done?

A:

Q: Consider `countdown(3)`. What will the recursive call look like (with the value instead of the actual argument expression)?

A:

Q: What part of the output is handled by the recursive call?

A:

Q: So what is the "little bit of work" that is done in `countdown(3)`'s recursive case?

A:

⟨ The Cat in the Hat Comes Back! (Go slow, only takes 10-15 minutes.) ⟩

The recursive case is often tricky! Two parts:

- Do a small amount of the work.
- Make the recursive call.

Always write the base case first. Then, if you get stuck on the recursive case, consider the case where the recursive call will call the base case. (For example, `countdown(1)`. Then think about what the "little bit of work" will be that your recursive case will do and put the two parts together. Often, this will get you close to writing your recursive case.

Example: I'm going to give the count for just this number, then trust that the recursive call will count down the rest.

```
>>> final_countdown(5)
5
4
3
2
1
Blastoff!
>>> final_countdown(20)
10
9
8
7
...
3
2
1
Blastoff!
```

Q: How could we implement `final_countdown`? It only counts down the last ten numbers.

A:

```
>>> count_to(10)
1
2
...
10
Done!
>>> count_to_helper(12, 15)
12
13
14
15
Done!
>>> count_to_helper(100, 4)
Done!
>>> count_to(3)
1
2
3
Done!
>>> evens_between(3, 7)
4
6
>>> evens_between(2, 5)
2
4
>>>
```


Q: What's the base case for `count_to_helper`?

A:

Q: Implement `count_to`! Hint: You'll need to implement `count_to_helper` first. Bonus challenge: `evens_between`!

A:

5.10 Keyboard Input

`input()` gets input while the program is running! It is a fruitful function!

```
>>> word = input()
Monkieeeez!
>>> print(word)
Monkieeeez!
```

You can add text and ask a question!


```
>>> name = input("What is your name?")
What is your name?Kyle Burke
>>> greet(name)
Hi, Kyle Burke!
```

Q: What's the problem here?

A:

One solution: add more spaces.

```
>>> name = input("What is your name?  ")
What is your name?  Kyle Burke
>>> greet(name)
Hi, Kyle Burke!
```

Another solution: add a line break!

```
>>> name = input("What is your name?\n")
What is your name?
Kyle Burke
>>> greet(name)
Hi, Kyle Burke!
```

Let's add `input` to the body of a function!

```
>>> greet_prompt()
Whom would you like to greet?
Steve
Hi, Steve!
```

Q: Implement it! Header:

```
def greet_prompt():
```

A:

```
>>> is_between_prompt(0, 10)
Which number would you like to test?
0
Yes, it is!
>>> is_between_prompt(0, 10)
Which number would you like to test?
16
No, it isn't!
```


Q:

Implement it! Header:

```
def is_between_prompt(lower, upper):
```

Hint: `input` always returns a string! Bonus Challenge: make it work so it always uses the min of inputs as lower and max as upper

A:


```
>>> is_between_maybe_keep_asking(-6, 3)
Which number would you like to test?
0
Yes, it is!
Would you like to test another number?
yes
Which number would you like to test?
24
No, it isn't!
Would you like to test another number?
no
>>> is_between_keep_asking(-1, 5)
Which number would you like to test?
0
Yes, it is!
Which number would you like to test?
24
No, it isn't!
...
```

Q: Implement it! Header:

```
def is_between_keep_asking(lower, upper):
```

Challenge: `is_between_maybe_keep_asking(lower, upper)`

A:

6 Fruitful Functions

6.1 Return Values

Alright, it's time to write fruitful functions! Now we will write functions that return a value! We've already seen:

```
>>> import math
>>> x = math.sin(0)
>>> print(x)
0.0
>>> x = greet('Kyle')
Hi, Kyle!
>>> print(x)
None
```

What if we want a function like `math.sin` that returns a value when called? We will use the keyword `return`.

```
def area_of_circle(radius):
    '''Returns the area of a circle with the
specified radius.'''
    area = math.pi * (radius ** 2)
    return area
```

```
>>> area = area_of_circle(3)
>>> print(area)
28.27433
>>> print(area_of_circle(5))
78.539...
```

Q: What does the `return area` line do?

A:

Q: What do you think `return` alone does?

A:

6.2 Incremental Development

6.3 Composition

I could use this to calculate the volume of a cylinder with radius 10 and height 7!

⟨ Draw a picture of cylinder and describe formula for the volume. ⟩

```
>>> circle_area = area_of_circle(10)
>>> volume = circle_area * 7
>>> print(volume)
2199.11
```

Notice: keyword `return`, followed by a value.

Q: How might we write the body of `area_of_circle` in one line?

A:

```
>>> area = area_of_triangle(base = 4, height = 5)
>>> print(area)
10.0
>>> trapezoid_area = area_of_trapezoid(top = 4,
base = 6, height = 3)
>>> print(trapezoid_area)
15.0
```

Q:

Implement it! Header:

```
area_of_triangle(base, height):
```

Challenge:

```
area_of_trapezoid(top, base, height)
```

A:

```
>>> volume = volume_of_cylinder(radius = 5, height
= 7)
>>> print(volume)
549.7787143782139
>>> cone_volume = volume_of_cone(base_radius = 7,
height = 18)
>>> print(cone_volume)
42.0
```

⟨ Draw a picture! ⟩

Q:

Implement it! Header:

```
def volume_of_cylinder(radius, height):
```

Hint: use `area_of_circle`
Challenge: `volume_of_cone`

A:

```
>>> number = absolute_value(4)
>>> number
4
>>> weight = absolute_value(-27)
>>> print weight
27
```

Q:

Write this! (There is a function that does it for you in the math package; I want you to try writing it on your own!)

```
def absolute_value(number):
```

Challenge: after you get it to work, do in one line?

A:

You have to be careful! Mine doesn't work for zero!

```
>>> print absolute_value(0)
None
```


6.4 Boolean Functions

Can write Boolean Fruitful functions as well!

```
>>> parity = is_even(3)
>>> print(parity)
False
>>> is_even(222)
True
>>> is_square(4)
True
>>> is_square(5)
False
>>>
```

Q:

Implement it! Header:

```
def is_even(number):
```

Hint: use modulus operator!
Challenge: write in one line. Bigger Challenge: write `is_square(number)`

A:

```
>>> parity = is_odd(3)
>>> print(parity)
True
>>> is_odd(222)
False
TODO: needs a bonus challenge!
```


Q: Implement `def is_odd(number):` Bonus challenge: composition — call `is_even`!

A:

```
>>> between = is_between(5, 10)
Which number would you like to test?
8
>>> print(between)
True
>>> test_value = is_between(5, 10)
Which number would you like to test?
265
>>> if test_value:
    print("That's crazy!")
else:
    print("That's what I expected.")
...
That's what I expected
>>> is_between(10, 5)
Which number would you like to test?
7
True
>>>
```


Q:

Implement it! Header:

```
def is_between(lower, upper):
```

Bonus Challenge: get it to work so that the first doesn't have to be the lower bound.

A:

Let's do a tough one! Let's write a function that calculates the area of a rectangle given two corners.

```
>>> area_of_rectangle_between_points(1, 5, 3, 1)
8
>>> area = area_of_rectangle_between_points(4, 6,
6, 4)
>>> print(area)
4
>>> area_of_triangle_in_points(0, 0, -3, 4, -8.5,
3)
12.5
```

⟨ Draw a picture! ⟩

Q: Implement it! Header:

```
def area_of_rectangle_between_points(x0, y0,
x1, y1):
```

Hint: give docstring. Bonus challenge: `area_of_triangle_in_points`

A:

Q: Could we rewrite this so that it doesn't matter which point is which, so long as they are in opposite corners?

A:

Q: What do we need to do now?

A:

6.4.1 Testing Fruitful Functions

Q: How were we testing functions that printed instead of returned?

A:

```
print('Testing greet(Elman):')
print('Should output:  Hi, Elman!')
print('Actual output:  ', end = '')
greet('Elman')
```

Q: Can we do the same sort of thing with fruitful functions?

A: Yes!

```
print('Testing absolute_value(35):')
print('Should output:  35')
print('Actual output:',
absolute_value(35))
print('')
print(''Testing
absolute_value(-3012):')
print('Should output:  3012')
print('Actual output:',
absolute_value(-3012))
print('')
print(''Testing absolute_value(0):')
print('Should output:  0')
print('Actual output:',
absolute_value(0))
print('')
```


Q: Can we do better?

A: Absolutely!

Q: Rewrite the three tests so each only prints one line instead of three. Hint: use conditionals.

A:

Q: Do you think it's more important to print out a message when a function works or when it's incorrect?

Q: Hmmm... Think back to one of our software design techniques. Which one could we use to improve our code here?

A: There's lots of repeated code, so... encapsulation!

Okay then, let's wrap that conditional up in a function. (Recall that my `absolute_value` function is broken for 0.)

```
>>> test_function('absolute_value',
absolute_value(35), 35)
>>> test_function('absolute_value',
absolute_value(-3012), 3012)
>>> test_function('absolute_value',
absolute_value(0), 0)
Error while testing the absolute_value function!
Returned None instead of 0
```

Q: Write `test_function(name, result, goal)`!

A:

6.5 Fruitful Recursion

Let's combine fruitful functions and recursion to perform calculations! What about factorial?

Q: How does factorial work?

$$n! = \begin{cases} 1 & , n = 0 \\ n \times ((n-1)!) & , n > 0 \end{cases}$$

```
>>> x = factorial(0)
>>> print x
1
>>> y = factorial(4)
>>> print y
24
>>> factorial(10)
...
```

Q: Implement it! This one's hard! Header:

```
def factorial(integer):
```

Hint: Base case first!

Hint: Two parts to recursive case!

A:

⟨ Draw a stack diagram for `>>> factorial(5)`

120. ⟩

6.6 Leap of Faith

Leap of faith is more important with fruitful recursion! Expect the recursive call to work!

Moving away from print, returning instead! But, `print` is still very useful in debugging!

⟨ Example: add a print statement to factorial:

```
print("Result of recursive call:  " +
str(recursive_result))
```

⟩

I have an even more comprehensive version with more prints.

```
>>> x = factorial_with_prints(4)
Calculating 4 recursively...
Calculating 3 recursively...
Calculating 2 recursively...
Calculating 1 recursively...
At the basecase:  0 = 1
1 is:  1 times 1 = 1
2 is:  2 times 1 = 2
3 is:  3 times 2 = 6
4 is:  4 times 6 = 24
>>> print x
24
```

If something were going wrong, I could tell here, possibly without even looking at my code! Here's an example:


```
>>> y = factorial_wrong(4)
Calculating 4 recursively...
Calculating 3 recursively...
Calculating 2 recursively...
Calculating 1 recursively...
At the basecase:  0 = 1
1 is:  1 times 1 = 2
2 is:  2 times 2 = 4
3 is:  3 times 4 = 7
4 is:  4 times 7 = 11
>>> print y
11
```

Notice, you can tell what I'm doing wrong without seeing this code!

6.7 Example: Fibonacci

Q: What are the first five numbers in the Fibonacci sequence?

A:

Q: How do you calculate the next number from the previous two?

A:

⟨ Draw out a bunch of the sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... ⟩


```
>>> f = fibonacci(0)
>>> print(f)
0
>>> print(fibonacci(0), fibonacci(1),
fibonacci(2), fibonacci(3), fibonacci(4))
0 1 1 2 3
>>> fibonacci(10)
55
>>> fibonacci(20)
6765
>>> fibonacci(40) #Wait for it...
102334155
>>> print(tribonacci(0), tribonacci(1),
tribonacci(2), tribonacci(3), tribonacci(4),
tribonacci(5))
0 1 1 2 4 7
>>> x = tribonacci(10)
>>> print(x)
149
```

Q: Implement it! Again, hard! Header:

```
def fibonacci(index):
```

Hint: Two base cases!
Hint: Two recursive calls!

A:

⟨ Draw stack diagram for `>>> fibonacci(5)`
5. ⟩

Notice this goes pretty slowly... I have a faster version!

```
>>> fast_fibonacci(40)
102334155
>>> fast_fibonacci(100)
354224848179261915075L
```

I'm not going to ask you to do this, but I did it using a helper function that keeps track of two values at a time. You can return two values using tuples, which we will learn about in a future chapter.

Explain Triangular[7] and Tetrahedral numbers[8]!

```
>>> triangular_number(3)
6
>>> triangular_number(5)
15
>>> triangular_number(100)
5050
>>> tetrahedral_number(2)
4
>>> tetrahedral_number(5)
35
```

[7]More info about Triangular numbers: `https://en.wikipedia.org/wiki/Triangular_number`.
[8]More info about Tetrahedral numbers: `https://en.wikipedia.org/wiki/Tetrahedral_number`

Q: Implement `triangular_number`! (Do it recursively!) Bonus challenge: `tetrahedral_number`.

A:

6.8 Checking Types and Guardians

Q: What happens if I try to call

```
fibonacci(-1)
```

with the code we've written?

A:

Let's see what happens when I run the code *I've* written...

```
>>> fibonacci(-1)
Fibonacci is not defined for negative numbers!
```


Q: Wow! How did I make that happen?

A: I added a *guardian*, a test in a function to handle inappropriate inputs. Here's my guardian:

```
if index < 0:
    print("Fibonacci is not defined
for negative numbers!")
elif ...
```

Q: There's a name for the practice of protecting code from unintended situations. What is it?

A: *Defensive Programming*

Q: What else might work to break `fibonacci`?

A: A float! Let's try it out!

```
>>> fibonacci(2.5)
fibonacci is only defined for integers!
>>> fibonacci("monkeyface")
fibonacci is only defined for integers!
```

Whoa! How can we make that work? Use: `isinstance`!


```
>>> is_int = isinstance(5, int)
>>> is_int
True
>>> not_int = isinstance("monkey", int)
>>> print(not_int)
False
>>> isinstance(7, float)
False
>>> isinstance("apricot", str)
True
```

Q: Implement this guardian! Bonus challenge: implement it to also cover the following cases:

```
>>> fibonacci(10.0)
55
```

A:

Q: Does the order of these matter?

A:

Q: What is being returned when the body of these guardians is run?

A:

We can add returns to unnest rest of the function:

```
(On Board)
if not isinstance(index, int):
    print "Fibonacci is only defined for
integers!"
    return
elif index < 0:
    print "Fibonacci is not defined for negative
numbers!"
    return
...rest of the body...
```

Q: Do we think that checking whether a value is an integer is something we might want to do often?

A: Maybe, yeah!


```
>>> is_integer(53)
True
>>> x = is_integer('monkey')
>>> print(x)
False
>>> y = is_integer(-13.0)
>>> print(y)
True
>>> z = is_natural(-13)
>>> z
False
>>> is_natural(10)
True
>>> is_natural(10.0)
True
```

Q: Implement it! Header: `def is_integer(value):`! Bonus challenge: `is_natural`.

A:

Q: What happens if you call `fibonacci` on a boolean value? It should break, right?

A: Crazy stuff happens!

```
>>> fibonacci(True)
1 (Yours says True, because of base case(s).)
>>> fibonacci(False)
0 (False)
```

Q: What's happening here?

A: Python treats booleans as integers! Here are some more crazy cases!

```
>>> False == 0
True
>>> True == 1
True
>>> 5 + True
6
```

7 Iteration

We've learned how to write code to answer any question a computer can answer. Are we done? No! There are lots more tools that can make things easier.

7.1 Multiple Assignment

Something weird:


```
>>> x = 5
>>> print(x)
5
>>> x = 100
>>> print(x)
100
```

Multiple Assignment: Same variable, but different values at different times!

```
>>> x = 5
>>> x = x + 1
>>> print(x)
6
```

7.2 Updating Variables

Variable Updating: new value of the variable depends on the old value.

```
>>> for i in range(10):
    x = x + 1
...
>>> print(x)
16
```

Def: *Increment* Updating a variable by adding 1.

Def: *Decrement* Updating a variable by subtracting 1.

One of the most important things we use computers for is to perform repetitive calculations. They're good at it; we're not.

7.3 `while`

Def: *Iteration* Repetitive calculation is called iteration.

```
>>> def countdown_alternative(n):
    while n > 0:
        print(n)
        n = n-1
    print("Blastoff!")
...
>>>
```

⟨ Describe `while`. ⟩

Q: What will happen when I call: `countdown_alternative`?

```
>>> countdown_alternative(5)
5
4
3
2
1
Blastoff!
```

Def: *Loop* Loops back through a block of code multiple times. `while` is a combination of `if` and `for`.

The body of a loop should change variables so that eventually the condition becomes false... otherwise: *infinite loop*!

```
>>> x = 7
>>> y = 0
>>> while x > 5:
    y = y + 1
    print("y = ", y)
...  (Don't hit enter yet!)
```

Q: What will happen when I hit enter?

A:

⟨ Do it! ⟩

Q: Where is there an infinite loop in your everyday life? Hint: something you might do every morning. (Maybe not now that you're in college...)

A:

Sometimes it's not clear whether there's an infinite loop. It's not obvious whether a program will finish, or *terminate*. Look at page 65[9]: `sequence(n)`.

```
def sequence(n):
    '''Prints out the Collatz sequence starting
with n.'''
    while n != 1:
        print(n, end = " ")
        if n % 2 == 0:
            #n is even
            n = n / 2
        else:
            #n is odd
            n = n*3 + 1
```

Q: Type it in! Will it always terminate?

```
>>> sequence(3)
3 10 5 16 8 4 2
```

Q: Is there a positive integer for `x` for which this doesn't terminate?

A:

[9] http://greenteapress.com/thinkpython2/thinkpython2.pdf#page=87

```
>>> print_monkeys(2)
monkey
monkey
>>> print_monkeys(4)
monkey
monkey
monkey
monkey
>>> print_monkey_mountain(4)
monkey
monkeymonkey
monkeymonkeymonkey
monkeymonkeymonkeymonkey
```

TODO: switch to monkeys

Q: Implement it using iteration! Header:

```
def print_monkeys(n):
```

Hint: use a `while` loop based on
`print_monkey_mountain`.

```
>>> print_fibonacci_sequence(5)
0 1 1 2 3
>>> print_fibonacci_sequence(30)
0 1 1 2 3 5 8 13 21 34 ...
>>> fast_print_fibonacci_sequence(100)
...  much faster!...
```


Q: Implement it! Header:
`def print_fibonacci_sequence(n):`
Bonus challenge: `fast_print_fibonacci_sequence`

A:

```
>>> multiply(3, 21)
63
>>> x = multiply(4, 5)
>>> print(x)
20
>>> multiply(0, 5)
0
>>> exponentiate(57, 0)
1
>>> bigNumber = exponentiate(5, 3)
>>> print bigNumber
125
```


Q: Implement `multiply` without using `*`. Bonus challenge: implement `exponentiate` without using `**`.
Hint: give the first line
double hint: give the last line

A:

7.4 break

We can also use `break` to escape a loop anywhere instead of only at the top. Try out example on bottom of page 66[10]. I encapsulated the example into a function:

```
>>> input_break_test()
> monkey
monkey
> done
All done!
>>>
```

7.5 Square Roots

TODO: I need a bonus challenge problem for this section. Taylor expansion or something like that?

Calculating square roots can be hard. Let's pretend we didn't have a nice operation to do it. There is a method to improve guesses: *Newton's Method*.

[10] `http://greenteapress.com/thinkpython2/thinkpython2.pdf#page=88`

If you have a guess for $\sqrt{a}$, there is a way to improve that guess dramatically. Start with estimate: x. Then a better estimate is: $\frac{x+\frac{a}{x}}{2}$.

Let's test this out in Python.

```
>>> a = 4.0
>>> guess = 3.0
>>> guess ** 2
9.0
>>> better_guess = (guess + a/guess)/2
>>> print(better_guess)
2.16666666666665
>>> print(better_guess ** 2)
????
```

That is much closer to 4 than 9 was!

Q: Can we do better?

A:

```
>>> guess = better_guess
>>> better_guess = (guess + a/guess)/2
>>> print(better_guess)
2.00641025641
>>> print(better_guess ** 2)
```

Again!

```
>>> guess = better_guess
>>> better_guess = (guess + a/guess)/2
>>> print(better_guess ** 2)
??
>>> guess = better_guess
>>> better_guess = (guess + a/guess)/2
>>> print(better_guess ** 2)
??
```


Q: It could take a while to get to the correct answer. How do we know when we're there?

A:

```
>>> guess = better_guess
>>> better_guess = (guess + a/guess)/2
>>> print(better_guess ** 2)
4.0
>>> print(better_guess)
2.0
>>> guess = better_guess
>>> better_guess = (guess + a/guess)/2
>>> print(better_guess)
2.0
```

```
>>> square_root_from_guess(4.0, 3.0)
Given guess:  3.0
Better guess:  2.1666666666667
Better guess:  ...
Better guess:  2.0
Better guess:  2.0
Best guess:  2.0
2.0
>>> root = square_root_from_guess(16.0, 10.0)
Given guess:  10.0
Better guess:  ...
Best guess:  4.0
>>> print(root)
4.0
```


Q: Implement it! Header:

```
def square_root_from_guess(base, guess):
```

(Don't use `math.sqrt`!)

A:

This can be a bit dangerous! What if the code doesn't do a good job approximating and keeps being close, but not exact?

```
>>> root = square_root_from_guess(6.0, 3.0)
Given guess:  3.0
Better guess:  ...
...
(infinite loop.  Mine works, but theirs won't.)
```

```
>>> root = math.sqrt(6.0)
>>> root ** 2
5.999999999999999991
```

Q: How can we fix `square_root_from_guess` to make this work?

A:


```
def square_root_from_guess(base, guess, epsilon):
    print('Base guess:', guess)
    while abs(base - (guess ** 2)) > epsilon:
        better_guess = (guess + base/guess)/2
        print('Better Guess:', better_guess)
        guess = better_guess
    return better_guess
```

Notice that mine works whether or not I specify the `epsilon` parameter:

```
>>> square_root_from_guess(4.0, 3.0)
2.0
>>> square_root_from_guess(4.0, 3.0, epsilon =
.001)
2.0001something
```

Q: How did I do this?

A:

Q: Challenge: change `epsilon` to be a default parameter.

A:

Q: What's an important part of default parameters?

A: Can't have a non-default param after default ones. All defaults at the end of the param list.


```
>>> my_square_root(15)
3.872983346207417
>>> root = my_square_root(13)
>>> print(root)
3.6055512754639896
>>> root ** 2
13.000000000000002
```

Q: Challenge: write `my_square_root`. Hint: two parts:

- Remove scaffolding.
- Add the wrapper function.

A:

Q: What is this called, when you have a function that just calls another function with more parameters?

A: *Worker-Wrapper* programming pattern.

7.6 Algorithms

Newton's method is an example of an algorithm.

Def: *algorithm* A mechanical process for solving a category of problems.

You already know lots of algorithms!

- how to add two numbers with any amount of digits
- how to count the number of branches in a pile of sticks

Some things are not algorithmic: adding *single-digit* numbers. You probably memorized all combinations!

Carrying out algorithm doesn't require intelligence. Just follow steps.

Creating algorithm does! Central to the art of programming!

Q: How do we approximate π?

A: Algorithms! Example: Gauss-Legendre Algorithm[11]

Q: How do we use these formulas to create an algorithm?

A: Start with:

- $a_0 = 1$
- $b_0 = 1/\sqrt{2}$
- $t_0 = 1/4$
- $p_0 = 1$

Our first approximation is:

$$\pi \approx \frac{(a_0 + b_0)^2}{4t_0} \approx 2.914$$

Q: That's not too great. How can we improve it? Say we have a_k, b_k, t_k, and p_k, and they provide an approximation using the same formula:

$$\frac{(a_k + b_k)^2}{4t_k}$$

How can we find the next round: a_{k+1}, b_{k+1}, t_{k+1}, and p_{k+1} that will produce a better approximation? (I'm not expecting you to know this without looking up the Wikipedia page for Gauss-Legendre.)

A:

- $a_{k+1} = (a_k + b_k)/2$
- $b_{k+1} = \sqrt{a_k \cdot b_k}$
- $t_{k+1} = t_k - p_k(a_k - a_{k+1})^2$
- $p_{k+1} = 2p_k$

TODO: talk about the rate of the number of correct digits.

```
>>> bad_pi = my_pi(0)
>>> print(bad_pi)
2.914213562373095
>>> my_pi(1)
3.1405792505221686
>>> my_pi(2)
3.141592646213543
>>> my_pi(3)
3.141592653589794
>>> my_pi(10)
...
```


Q: Implement `my_pi`! Hint: outside the while loop I created variables `a`, `b`, `t`, and `p`. Inside, I created variables for the next round: `a_next`, `b_next`, etc. After setting those four new variables, I reset the original 4.
Bonus challenge: Look up another π approximation to use!

A:

7.7 Debugging

Debugging! If I have a program with 100 lines that has a semantic error, but I don't know where the error is occuring, I have a few options to debug, using print:

- Put a bunch of print statements inbetween as many lines as possible (100?)
- Put just one print statement in and see if the program makes sense up to that point.

Q: If I do that second option, where should that print statement go?

A:

Rules out biggest part of code where first error is occurring!

8 Strings

We did something weird with `is_palindrome` in a recent project...

```
>>> fruit = "banana"
>>> letter = fruit[1]
```

8.1 Strings as Sequences

Q: What just happened? What are those brackets?

A: Consider string as a sequence of characters. Can use brackets to access individual letters. Expression in brackets: *index*. Indicates which character you want!

```
>>> print(letter)
a
```


Q: Why did that happen?

A:

```
>>> fruit[0]
a
```

"Zeroeth", "Oneth", "Twoeth", "Threeeth", "Foureth", ...

Q: What will happen if I try to use a float as the index?

A:

```
>>> fruit[3.14]
Type Error!
```

8.2 len

```
>>> length = len(fruit)
>>> print(length)
6
```

Q: What will happen if I try

```
last = fruit[length]
```

?

A:

```
>>> last = fruit[length]
IndexError!
```


Since we started with 0, 6 is too high! Two options.

```
>>> last = fruit[length - 1]
>>> print(last)
a
>>> fruit[-1]
'a'
```

Q: Which `'a'` is this?

A:

Can either index from $[0, length - 1]$ or $[-length, -1]$.

```
>>> middle_char('apple')
'p'
>>> char = middle_char('banana')
>>> print(char)
n
>>> x = first_middle_last('apple')
>>> print(x)
'ape'
```

Q: Write `middle_char`! Bonus challenge: `first_middle_last`.

A:

TODO: change this to last_characters with middle_chars as the bonus challenge. (first_characters is a project problem).


```
>>> first_characters('beluga', 3)
'bel'
>>> prefix = first_characters('onomatopoeia', 6)
>>> print(prefix)
onomat
>>> first_characters('skullduggery', 20)
skullduggery
>>> last_characters('banana', 5)
'anana'
```

Q: Code up `first_characters`. Bonus challenge: `last_characters`.

A:

```
>>> extend_string('beluga', 15)
'belugabelugabelugabel'
>>> s = extend_string('giraffe', 4)
>>> print(s)
gira
>>> x = pongify("banana")
>>> print(x)
bananaananabbanana
>>> pongify('ape')
'apeepaape'
```


Q: Write `extend_string`! Hint: I used recursion, though you can also do iteration. Bonus challenge: `pongify`.

A:

```
>>> middle_chars("antelope hoard", 6)
'lope h'
>>> middle = middle_chars("terrasque", 3)
>>> print('middle:', middle)
middle:  ras
```


Q: Code up `middle_chars`. TODO: make a bonus challenge for this.

A:

8.3 Traversing with `for`

Q: What if we want to do something to each letter in a string?

```
>>> index = 0
>>> fruit = 'banana'
>>> while index < len(fruit):
    character = fruit[index]
    print(character)
    index = index + 1

b
a
n
a
n
a
>>>
```

This loop *traverses* the string, printing each letter.

Def: *traverse* To visit each element in a data collection.

```
>>> forward_print("monkey")
m
o
n
k
e
y
```

Q: Implement it! Header:

```
def forward_print(string):
```

Hint: encapsulate the previous code.

```
>>> backward_print("monkey")
y
e
k
n
o
m
>>> primate = "primeape"
>>> backward_print(primate)
e
...
p
```


Q: Implement it! Header:

```
def backward_print(string)
```

A:

This sort of thing happens a lot, so Python has a way to write it more simply.

```
>>> animal = "mankey"
>>> for character in animal:
    print(character)

m
a
n
k
e
y
```

Whoa! `for` without `range`!

Q: How does this work?

A:

Q: Rewrite `forward_print` using a `for` loop!

A:

Q: Which `for` loop is the following `while` loop equivalent to?

```
index = 0
while index < len(y):
    x = y[index]
    A
    B
    ⋮
    Q
    index += 1
```

A:

There are even more uses of `for`, which we will see soon!
Look at the example on page 73[12]: Make Way for Ducklings!
I encapsulated this into a function!

[12] `http://greenteapress.com/thinkpython2/thinkpython2.pdf#page=95`

```
>>> print_duckling_names('JKLMNOPQ')
Jack
Kack
Lack
Mack
Nack
Ouack
Pack
Quack
>>> print_duckling_names('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
...
```

Q: Implement it! Header: `print_duckling_names(prefixes)`. It's okay if the `u`'s don't show up. Bonus challenge: get the `u`'s to print after `A`, `O`, and `Q`.

A:

```
>>> has_vowels('onomatopoeia')
True
>>> b = has_vowels('styx')
>>> if not b:
...     print('crazy!')

crazy!
>>> vowels = just_vowels('Sheep go to Heaven')
>>> print(vowels)
eeooeae
>>> just_vowels('onomatopoeia')
'ooaooeia'
```


Q: Implement `has_vowels(string)`. Bonus challenge: `just_vowels`.

A:

8.4 String Slices

Sometimes we want to look at just a piece of a string!

```
>>> word = 'smiles'
>>> subword = word[0:5]
>>> print(subword)
smile
>>> subsubword = word[1:5]
>>> print(subsubword)
mile
>>> print(word[1:5])
mile
>>> print('smiles'[1:5])
mile
```

string[n:m] means the chunk or *slice* of the string from (and including) the nth character to (but not including) the mth character.

Helpful: consider the indices occurring between characters.

TODO: draw picture similar to page 73[13], but using 'monkey'.

[13] http://greenteapress.com/thinkpython2/thinkpython2.pdf#page=95

```
>>> animal = "monkey"
>>> print(animal[2:5])
nke
```

Q: What is the length of slice: `string[n:m]`?

A:

```
>>> food = 'hamburger'
>>> food[i1:i2]
'urge'
```

Q: I set `i1, i2` prior to class. What are their values?

A:

```
>>> animal = 'monkey'
>>> animal[i3:i4]
'monk'
>>> animal[i5:i6]
'key'
```

Q: What about `i3, i4, i5, i6`?

A:

We can actually drop the indices for the last two!


```
>>> print(animal[:4])
monk
>>> print(animal[3:])
key
```

Q: What will happen with

```
>>> print(animal[:])
```

?

A:

Q: What will happen with

```
>>> animal[4:4]
```

?

A:

Q: What about

```
>>> animal[4:3]
```

?

A:


```
>>> for i in range(4):
...       print('i:', i)
...
i:  0
i:  1
i:  2
i:  3
>>> print_start_slices_of('monkey')
m
mo
mon
monk
monke
monkey
>>> print_start_slices_of('I')
I
>>> print_end_slices_of('ape')
e
pe
ape
```

Q: Implement it! Header:

```
print_start_slices_of(string):
```

Hint: use a loop!
Bonus challenge: `print_end_slices_of`

A:

```
>>> starts_with('blastoise', 'blast')
True
>>> nope = starts_with('pikachu', 'pichu')
>>> print(nope)
False
>>> starts_with('abra', 'abra')
True
>>> starts_with('abra', 'abrakadabra')
False
>>> ends_with('mankey', 'man')
False
>>> ends_with('mankey', 'key')
True
```

Q: Implement `starts_with(string, prefix)` Hint: can use a loop or slices. Bonus Challenge: `ends_with`

A:

8.5 Strings are Immutable

Strings are immutable!

```
>>> greeting = 'Hi, Stevey!'
>>> greeting[9] = 'n'
TypeError:  ....
```


Q: Why did that happen?

A:

You'd have to create a new string.

```
>>> new_greeting = greeting[:9] + 'n' +
greeting[10:]
>>> print(new_greeting)
Hi, Steven!
>>> print(greeting)
Hi, Stevey!
```

8.6 String Searching

Add `find` to your code, from page 74[14]:

```
(On Board)
def find(word, letter):
    LEAVE SPACE FOR DOCSTRING
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index += 1
    return -1
```

```
>>> find('abrakadabra', 'b')
1
```

[14] http://greenteapress.com/thinkpython2/thinkpython2.pdf#page=96

Q: What is a good docstring for `find`?

A:

```
>>> last_index = find_last('abrakadabra', 'a')
>>> print(last_index)
10
>>> find_last('monkey', 'z')
-1
>>> index = find_substring('charizard', 'riza')
>>> print(index)
3
```

Q: Write `find_last`! Hint: I wrote a solution that searches left-to-right. Bonus challenge: `find_substring`.

A:

8.7 Counting

```
(On Board)
def count(word, character):
    LEAVE SPACE FOR DOCSTRING LATER!
    count = 0
    for letter in word:
        if letter == character:
            count += 1
    return count
```

```
>>> count('abrakadabra', 'b')
2
```

Q: What would be a good docstring for `count`?

A:

```
>>> m = max_count('monkey')
>>> print(m)
1
>>> max_count('abrakadabra')
5
>>> char = most_frequent_character('abrakadabra')
>>> print(char)
a
```


Q: Try to write `max_count(string)`. Bonus challenge: `most_frequent_character`.

A:

8.8 String Methods

```
>>> word = 'monkey'
>>> new_word = word.upper()
>>> print(new_word)
MONKEY
>>> print(word)
monkey
```

`upper` is a *method* for strings.

Def: *method* A method is like a function, but specific to a data type.

The notation is similar to a function.

Idea: take the first argument, and move it to come infront of the function name. Use dot notation and you have a method!

```
>>> word = 'abrakadabra'
>>> index = word.find('r')
>>> print index
2
```

```
string.method_name(arguments)
```


Other types also have methods. We will learn how to use more, then later write our own!

To see all the string methods, type: `>>> help(str)`. Here are some examples:

```
>>> x = 'banana'.center(12)
>>> x
' banana '
>>> y = 'hi every monkey man!'
>>> z = y.title()
>>> z
'Hi Every Monkey Man!'
>>> z.replace('M', 'D')
'Hi Every Donkey Dan!'
>>> z.upper()
'HI EVERY MONKEY MAN!'
>>> to_fancy_title("monkey")
'~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ Monkey
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~'
>>> to_fancy_title("monkeys are awesome")
'~~~~~~~~~~~~~~~~~~~~~~~~ Monkeys Are Awesome
~~~~~~~~~~~~~~~~~~~~~~~~'
>>> cameled = snake_to_camel('snake_monkey')
>>> print(cameled)
snakeMonkey
>>> snake_to_camel('snake_monkey_banana_bear')
'snakeMonkeyBananaBear'
```


Q: Write `to_fancy_title(string)`. Hint: use the `title`, `replace`, and `center` methods. Bonus Challenge: `snake_to_camel`.

A:

8.9 in

We can use `in` to find substrings!

```
>>> 'Tea' in 'Team'
True
>>> 'I' in 'Team'
False
>>> 'me' in 'Team'
False
```

We've used `in` in for loops, but that use is a bit different. Code up `in_both` from page 76[15]:

```
(On Board):
def in_both(string_0, string_1):
    SPACE FOR DOCSTRING
    for letter in string_0:
        if letter in string_1:
            print(letter)
```

[15] http://greenteapress.com/thinkpython2/thinkpython2.pdf#page=98

```
>>> in_both('apples', 'oranges')
a
e
s
>>> in_both('love', 'hate')
e
```

Notice: `in` is used in two *very* different ways here!

- `for letter in string_0:` - Here, `in` is used with a for-loop for iteration.
- `if letter in string_1:` - Here, `in` is used to test inclusion. This part with `in` evaluates to a boolean.

Q:

Change `in_both` to be fruitful instead!

```
>>> x = in_both_f('apples', 'oranges')
>>> x
'aes'
>>> x = in_both_f('bananas', 'apples')
>>> x
'aaas'
```

Bonus Challenge: change `in_both` to work so that it doesn't include duplicates.

A:


```
 >>> in_all_three('apples', 'oranges', 'pears')
'aes'
>>> in_all_three('apples', 'oranges', 'bananas')
'as'
>>> in_all_three('bananas', 'oranges', 'apples')
'aaas'
```

Q: Implement it! Header:

```
in_all_three(word0, word1, word2):
```

Bonus challenge: get rid of repeats!

A:

```
>>> common = in_two_of_three('apples', 'oranges',
'pears')
>>> print('common:', common)
common:  appesr
>>> in_two_of_three('monkey', 'banana', 'apple')
'neaaa'
```


Q: Implement it! Header: `in_two_of_three(word0, word1, word2)`: Bonus challenge: get rid of repeats!

A:

```
>>> repeated = repeated_characters('Sheep go to
Heaven')
>>> print(repeated)
e o ee
>>> repeated_characters('eevee')
'eee'
```


Q: Implement `repeated_characters(string)`. Bonus challenge: get rid of repeats.

A:

8.10 String Comparisons

We have used == with strings. What about other comparators?

```
>>> 'apple' < 'banana'
True
>>> ''horse'' < ''monkey''
True
>>> ''orangutan'' < ''dolphin''
False
>>> 'monkey' < 'gorilla'
False
>>> ''monkey'' < ''monkey
False
>>> ''monkey'' <= ''monkey''
True
```


Q: What does < mean with strings?

A:

```
>>> 'apple' < 'Apple'
False
>>> 'Horse' < 'apple'
True
```

Q: What does this mean?

A:

```
>>> compare_to_banana('monkey')
Your word, monkey, comes after banana.
>>> compare_to_banana('apple')
Your word, apple, comes before banana.
>>> compare_to_banana('banana')
All right, bananas!
>>> compare_to_banana('Banana')
Your word, Banana, comes before banana.
```


Q: Implement it! Header:

```
def compare_to_banana(word):
```

A:

```
>>> first = earlier_string('abra', 'kadabra')
>>> print(first)
abra
>>> earlier_string('apple', 'abba')
'abba'
>>> strings_in_order('bulbasaur', 'ivysaur',
'venusaur')
True
>>> strings_in_order('abra', 'kadabra',
'alakazam')
False
```


Q: Implement `earlier_string(string_0, string_1)`.
Bonus challenge: `strings_in_order`.

A:

Q: When might this be useful?

A:

9 Files and Words

If we want a bunch of words, it may be easier to read those from a file.

9.1 Reading a File

⟨ Ask students to visit: the class schedule, find `words.txt`, and 'Save Target/File As...' to the Desktop. ⟩

```
>>> fileLocation = 'H:\\Desktop\words.txt'
>>> wordFile = open(fileLocation)
>>> print wordFile
<open file 'words.txt', mode 'r' ...
>>> wordFile.readline()
'aa\ n'
```


(aa is a kind of lava.)

Files keep track of where reading takes place. Try reading a line again!

```
>>> wordFile.readline()
'aah\ n'
>>> line = wordFile.readline()
>>> line
'aahed\ n'
>>> line = line.strip()
>>> line
'aahed'
```

```
>>> for x in wordFile:
    print x
(Don't hit enter twice!)
```

Q: What will happen when I hit enter?

A:

Q: How does a `for` loop work with a file? Example: `for x in wordFile:`

A:

9.2 Exercises

9.3 Searching

```
>>> printWordsContaining('ape')
...
```

Q: Implement it! Header:

```
def printWordsContaining(string):
```

Hint: use `in`!

A:

⟨ Talk about `.close()`. ⟩

```
>>> printWordsUsingAllOf('aeiou')
...
```

Q: Implement it! Warning: this one's hard! Header:

```
def printWordsUsingAllOf(characters):
```

(They probably won't finish this! Move on after a few.)

Maybe it would help if we defined a different function first!


```
>>> hasThemAll = usesAllOf(''monkeyjeans'',
''aeiou'')
>>> print hasThemAll
False
>>> usesAllOf(''monkeyjeans'', 'mno')
True
>>> usesNoneOf('monkeyjeans', 'zy')
False
>>> usesNoneOf('monkeyjeans', 'zt')
True
```

Q: Implement it! Header:

```
def usesAllOf(string, characters):
```

Bonus Challenge: `usesNoneOf`!

Q: Reimplement `printWordsUsingAll` to use `usesAll`. Bonus Challenge: `printWordsUsingNoneOf`

Q: What if I don't trust `words.txt` or want to use a different list? How can I generalize `printWordsUsingAll`?

A:

⟨ Ask students to download: `https://dl.dropbox.com/u/43416022/150/woerter.txt`. ⟩

```
>>> printWordsUsingAll('abcdefg',
'H:\\Desktop\woerter.txt')
...
```


Generalize that to match! Header:

Q:

```
printWordsUsingAll(characters,
fileLocation):
```

A:

Q: What if I still use words.txt as a default?

A:

Okay, let's do this the other way around. Let's write the function to test whether a word matches some criteria, then write a search to get all the matching words.

```
>>> leftHandWord(‘‘monkey’’)
False
>>> leftHandWord(’’trees’’)
True
```


Q: When does this function return `True`?

A:

Q: Implement it! Header:

```
def leftHandWord(word):
```

Bonus challenge: Implement `rightHandWord`.

```
>>> printLeftHandWords()
...
```

Q: Implement it! Header:

```
def printLeftHandWords():
```

Hint: use `leftHandWord` Bonus Challenge: `printRightHandWords()`

These all use the same searching "pattern": `for` loop traverses all the words in the file and a boolean function tests to see if the word matches.

```
>>> isAbecedarian(‘‘monkey’’)
False
>>> isAbecedarian(’’art’’)
True
```

Q: When does this function return `True`?

A:

Q:

Implement it! Header:

```
def isAbecedarian(word):
```

Progressive hints:

- use indices
- Can use iteration (while loop) or recursion! Whoa!
- 3 solutions on page 86[16].

```
>>> printAbecedarianWords()
...
```

Q:

Implement it! Header:

```
def printAbecedarianWords():
```

Bonus challenge: `printReverseAbecedarianWords()`

```
>>> printPalindromes()
aa
...
```

Q:

Implement it! Header:

```
def printPalindromes():
```

Hint: already did `isPalindrome()`

```
>>> isReversibleWord("radar")
True
>>> isReversibleWord("evil")
True
```


Q: Implement it! Header:

```
def isReversibleWord(word):
```

A:

```
>>> printReversibleWords()
...
```

Q: Implement it! Header:

```
def printReversibleWords():
```


Q: Why does this take so long?

A:

If there are n words, other searches take linear time (function in terms of n) but `printReversibleWords` takes quadratic time (function in terms of n^2).

There is a way to speed this up!

```
>>> speedyPrintReversibleWords()
...
```

You'll learn how to do that sort of stuff in Comp 250.

10 Lists

10.1 Lists are Sequences

Def: *list* A *list* is a data type that is a sequence of any values. Unlike a string, a list can contain more than characters.

Def: *elements* The values in a list are known as the *elements*.

```
>>> prime_list = [2, 3, 5, 7, 11, 13, 17]
>>> print(prime_list)
[2, 3, 5, 7, 11, 13, 17
>>> palindrome_list = ['a', 'I', 'radar',
'racecar']
>>> print(palindrome_list)
...  >>> type(palindrome_list)
<type 'list'>
```

Elements don't even need to be the same type!

```
>>> junx = [3.45, 'spam', 405, ['monkey', 5]]
```

Nested: a list within a list!


```
>>> empty = []
```

You can index lists just like strings.

```
>>> print(junx[1])
spam
>>> junx[1]
'spam'
```

10.2 Lists are Mutable

Unlike strings, lists *are* mutable!

```
>>> junx[1] = 'spamspamspam'
>>> print(junx)
[3.45, 'spamspamspam', 405, ['monkey', 5]]
```

Just like strings:

- any integer expression can be used as index
- `IndexError`s from non-existent elements
- `in` will test inclusion (but not sublists)

```
>>> 'spamspamspam' in junx
True
>>> 'banana' in junx
False
>>> ['monkey', 5] in junx
True
>>> [3.45, 'spamspamspam'] in junx
False
>>> 'monkey' in junx
False
```

```
>>> monkeys = ['tamarin', 'green monkey',
'macaque', 'marmoset', 'capuchin']
>>> swap_ends(monkeys)
>>> print(monkeys)
['capuchin', 'green monkey', 'macaque',
'marmoset', 'tamarin']
```


Q: Code up `swap_ends(t)`. Note: this function should be void. Bonus challenge: look up multiple assignment in Python and write the body in one line.

A:

10.3 Traversing a List

```
>>> songs = ['subdivisions', 'tom sawyer', 'the
trees', 'xanadu']
>>> index = 0
>>> while index < len(songs):
    print(songs[index])
    index += 1
...
subdivisions
...
>>> for song in songs:
    print(song)
...
subdivisions
...
>>> for song in songs:
    print(len(song))
...
12
...
```

Let's do an example modifying the list in the loop!


```
>>> import math
>>> numbers = [.5, -1, 8, math.pi, 1337]
>>> index = 0
>>> while index < len(numbers):
    numbers[index] = 2 * numbers[index]
    index += 1
...
>>> print(numbers)
[1.0, -2, 16, 6.28..., 2674]
```

Q: Can we use a `for` loop here?

A:

```
>>> for number in numbers:
    number = 3 * number
...
>>> print(numbers)
[1.0, -2, 16, 6.28..., 2674]
```

Didn't change from last time! However...

```
>>> for i in range(len(numbers)):
    numbers[i] = 3 * numbers[i]
...
>>> print(numbers)
```

Wow, that worked! Wha-? Why? How?

```
TODO: this no longer works in Python 3.  What to
say instead?
>>> range(len(numbers))
[0, 1, 2, 3, 4]
```

Q: What does the loop look like that would put the list back to normal? Write it!

A:

```
>>> stuff = [math.pi, 42, 'Neo', True, 0.01, [5,
10], math.sqrt]
>>> to_strings(stuff)
>>> print(stuff)
['3.14159265358979', '42', 'Neo', 'True', '0.01',
'[5, 10]', '<built-in function sqrt>']
>>> pokemon = ['bulbasaur', 'squirtle',
'charmander']
>>> to_mankeys(pokemon)
>>> print(pokemon)
['mankey', 'mankey', 'mankey']
>>> junk = [['blah', 50], 'blah', ['blah', [5]]]
>>> to_mankeys(junk)
>>> print(junk)
[['mankey', 'mankey'], 'mankey', ['mankey',
['mankey']]]
```

Q: Write `to_strings(t)`. Bonus challenge: `to_mankeys`. Note that this goes as deep as necessary! Hint: recursion on list elements.

10.4 List Operations

There are other similarities to strings!


```
>>> x = [1]
>>> x = x + [2, 3]
>>> print(x)
[1, 2, 3]
>>> x = x * 3
>>> print(x)
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> get_ones(5)
[1, 1, 1, 1, 1]
>>> three_ones = get_ones(3)
>>> print(three_ones)
[1, 1, 1]
>>> evens = get_evens(6)
>>> print(evens)
[0, 2, -2, 4, -4, 6]
>>> get_evens(0)
[]
```

Q: Code `get_ones(length)`. Bonus challenge: `get_evens`.

A:


```
>>> numbers = [4, 5, 6, 7]
>>> backwards = get_reverse_of(numbers)
>>> print('backwards:', backwards)
backwards:  [7, 6, 5, 4]
>>> print('numbers:', numbers)
numbers:  [4, 5, 6, 7]
>>> reverse_elements(numbers)
>>> print('numbers:', numbers)
numbers:  [7, 6, 5, 4]
```

Q: Code `get_reverse_of(t)`. Note: this doesn't modify the parameter. Hint: you need to use the most important list, the empty list! Bonus challenge: `reverse_elements`.

A:

10.5 List Slices

```
>>> x = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> y = x[2:7]
>>> print(y)
[3, 1, 2, 3, 1]
```

Remember, lists are mutable. So we can do this...

```
>>> y[1:4] = [2.5, 2, 1.5]
>>> print(y)
[3, 2.5, 2, 1.5, 1]
```



```
>>> reverse_middle(y)
>>> print(y)
[3, 1.5, 2, 2.5, 1]
>>> monotrenes = ['platypus', 'echidna']
>>> reverse_middle(monotrenes)
>>> print(monotrenes)
['platypus', 'echidna']
>>> shift_elements(y)
>>> print(y)
[2.5, 2, 1.5, 1, 3]
>>> shift_elements(y)
>>> print(y)
[2, 1.5, 1, 3, 2.5]
```

Q: Code `reverse_middle(t)`. Bonus challenge: `shift_elements`.

A:

We can include a third part to slices!

```
>>> primates = ['tamarin', 'lemur', 'orangutan',
'chimpanzee', 'gorilla']
>>> primates[::2]
['tamarin', 'orangutan', 'gorilla']
>>> primates[::-2]
['gorilla', 'orangutan', 'tamarin']
```


Q: Let's go back to `reverse_elements` and try writing this using *in one line* using slices. (If you've already done it, save that version and try to add this version.)

A:

10.6 List Methods

lists also have methods!

```
>>> x.append(.5)
>>> print(x)
[3, 2.5, 2, 1.5, 1, .5]
>>> x2 = [0, 3.5]
>>> x.extend(x2)
>>> print(x)
[3, 2.5, 2, 1.5, 1, .5, 0, 3.5]
>>> x.sort()
>>> print(x)
[0, .5, 1, 1.5, 2, 2.5, 3, 3.5]
```

Q: What is different about these methods as compared to string methods?

A:

Q: Are they fruitful?

A:


```
>>> numbers = [1, 9, 5, 5, 3, 7, 8, 2]
>>> reverse_sort(numbers)
>>> print(numbers)
[9, 8, 7, 5, 5, 3, 1]
>>> sorted = get_sorted_copy(numbers)
>>> print('sorted:', sorted)
sorted:  [1, 3, 5, 5, 7, 8, 9]
>>> print('numbers:', numbers)
numbers:  [9, 8, 7, 5, 5, 3, 1]
```

Q: Code `reverse_sort(t)`. Bonus challenge: `get_sorted_version`.

A:

```
>>> double_list(numbers)
>>> print(numbers)
[9, 8, 7, 5, 5, 3, 1, 9, 8, 7, 5, 5, 3, 1]
>>> double_list(numbers)
>>> print(numbers)
[9, 8, 7, 5, 5, 3, 1, 9, 8, 7, 5, 5, 3, 1, 9, 8,
7, 5, 5, 3, 1, 9, 8, 7, 5, 5, 3, 1]
>>> other_numbers = [2.4, 1.21, 62, 51.33]
>>> up_and_down(other_numbers)
>>> print(other_numbers)
[1.21, 2.4, 51.33, 62, 62, 51.33, 2.4, 1.21]
```


Q: Code up `double_list(t)`. Bonus challenge: `up_and_down`.

A:

10.7 Map, Filter, and Reduce

Often we want to sum a list.

```
>>> total = list_sum([6, 7, 8])
>>> print(total)
21
>>> list_sum(x)
14
>>> all_even([6, 7, 8])
False
>>> integers = [12, 24, 36, 60, 4, 2]
>>> all_even(integers)
True
```

Q: Implement it! Header:

```
def list_sum(numbers):
```

Hint: use an accumulator variable, starting at 0. Bonus challenge: `all_even`.

A:

Def: *reduce operation* This sort of thing is called a reduce operation. A *reduce operation* takes a list and returns a single value.

```
>>> more_numbers = [1, 5, 88, -4, 9, -999, 1]
>>> greatest = list_max(more_numbers)
>>> print(greatest)
88
>>> list_max(x)
3.5
>>> greatest_absolute = list_max_absolute(more_numbers)
>>> print(greatest_absolute)
-999
```

Q: Are these both reduces?

A:

Q: Implement it! Header:

```
def list_max(numbers):
```

Bonus Challenge: implement `list_max_absolute`.

A:


```
>>> exclamations = ['Bonus!', 'BOOyeah!',
'suhWEET!', 'WOOT!']
>>> little = all_lower_case(exclamations)
>>> print(little)
['bonus!', 'booyeah!', 'suhweet!', 'wOOt!']
>>> print(exclamations)
['Bonus!', 'BOOyeah!', 'suhWEET!', 'WOOT!']
>>> pokemon = ['bulbasaur', 'ivysaur',
'venusaur', 'squirtle']
>>> abbreviate_and_sort(pokemon)
['bulb', 'ivys', 'squi', 'venus']
```

Q: Is this a reduce operation?

A:

Q: Implement `all_lower_case(strings)` Hint: use `lower()` string method. Bonus challenge: `abbreviate_and_sort`.

A:

This sort of operation is a *map*.

Def: *map operation* A map operation does something to each element in the list.

This does not need to be fruitful!


```
>>> odds = [1, 3, 5, 7, 9]
>>> add_five_to_all(odds)
>>> print(odds)
[6, 8, 10, 12, 14]
>>> round_up_to_squares(odds)
>>> print(odds)
[9, 9, 16, 16, 16]
```

Q: Implement it! Header: `add_five_to_all(numbers)` Hint: this function is void!

A:

That one modifies the list instead of being fruitful. Hmm...

```
>>> integers = [0, -5, -3, 22, 1337, 42]
>>> odds = only_odds(integers)
>>> print(odds)
[-5, -3, 1337]
>>> print(integers)
[0, -5, -3, 22, 1337, 42]
>>> split_evens_and_odds([1, 2, 3, 4, 5])
[[2, 4], [1, 3, 5]]
```

Q: Is this a map operation?

A:

Def: *filter operation* A filter operation "filters out" some elements of the list.

Q: Implement it! Header: `only_odds(integers)` Hint: creates a new list. Bonus challenge: `split_evens_and_odds`. Double bonus challenge: write `splite_evens_and_odds` in only 4 lines.

A:

```
>>> caps = only_caps(exclamations)
>>> print(caps)
['WOOT!']
>>> only_caps(['I', 'LOVE', 'monkeys'])
['I', 'LOVE']
>>> caps_from_each(exclamations)
['B!', 'BOO!', 'WEET!', 'WOOT!']
```

Q: Is this also a filter?

A:

Q: Implement `only_caps(strings)`. Hint: creates a new list. Challenge: `caps_from_each`

A:

10.8 Deleting Elements

Let's remove the negative values from our list.

```
>>> integers = [0, -5, -3, 22, 1337, 42]
>>> x = integers.pop(1)
>>> print(integers)
[0, -3, 22, 1337, 42]
>>> print(x)
-5
>>> x = integers.pop(1)
>>> print(integers)
[0, 22, 1337, 42]
>>> print(x)
-3
```

Q: What is `pop`?

A:

We can also use `del` to remove elements.


```
>>> del integers[0]
>>> print(integers)
[22, 1337, 42]
>>> m = del integers[2]
Error!
>>> print(m)
NameError:  name 'm' is not defined
>>> del integers[2]
>>> print(integers)
[22, 1337]
```

A `del` statement is a special kind of statement. It does not return a value. We can also use it on a slice!

```
>>> letters = ['r', 'e', 's', 'p', 'e', 'c', 't']
>>> del letters[3:6]
>>> print(letters)
['r', 'e', 's', 't']
```

If we want to remove an element but don't know the element, we can use the `remove` method!

```
>>> letters = ['r', 'e', 's', 'p', 'e', 'c', 't']
>>> letters.remove('s')
>>> print letters
['r', 'e', 'p', 'e', 'c', 't']
>>> letters.remove('e')
>>> print letters
['r', 'p', 'e', 'c', 't']
```

With these tools, we can write void filters!


```
>>> integers = [0, -5, -3, 22, 1337, 42]
>>> remove_evens(integers)
>>> print(integers)
[0, -5, -3, 1337]
>>> x = [2, 4, 6, 8, 10, 12, 14]
>>> remove_evens(x)
>>> x
[]
>>> integers = [0, -5, -3, 22, 1337, 42]
>>> shrinkify(integers)
>>> print(integers)
[-2.5, -1.5, 11, 668.5, 21]
>>> shrinkify(integers)
>>> print(integers)
[-1.25, 5.5, 334.25, 10.5]
```

Q: Code `remove_evens(integers)`. Hint: a for loop won't work, you need to use a while! Bonus challenge: `shrinkify`.

A:

10.9 Lists and Strings

There are some cute things we can do with lists and strings!


```
>>> word = "subdivisions"
>>> letters = list(word)
>>> print(letters)
['s', 'u', ..., 's']
>>> string = "a farewell to kings"
>>> words = string.split()
>>> print(words)
['a', 'farewell', 'to', 'kings']
```

We can use a different delimiter!

```
>>> hyphenated = "crazy-angry-monkeys"
>>> hyphenated.split()
['crazy-angry-monkeys']
>>> hyphenated.split('-')
['crazy', 'angry', 'monkeys']
```

And we can rejoin strings!

```
>>> delimiter = ' '
>>> delimiter.join(words)
'a farewell to kings'
```

Let's write some functions with these methods!

```
>>> phrase = "my mom said that I need to eat
seventeen apples"
>>> sort_words(phrase)
'I apples eat mom my need said seventeen that to'
>>> sorted = sort_words("electrode diglett nidoran
mankey venusaur rattata fearow pidgey")
>>> print(sorted)
'diglett electrode fearow mankey nidoran pidgey
venusaur'
>>> sort_words_by_length(phrase)
'I my to mom eat said that need apples seventeen'
```


Q: Code `sort_words(string)`. Bonus challenge: `sort_words_by_length`. (Really hard! I used two nested while loops!)

10.10 Objects and Values

```
>>> a = 'monkey'
>>> b = 'monkey'
>>> a is b
True
```

Q: What might `is` mean?

A:

⟨ Draw pointer picture! ⟩
This doesn't work with all objects.

```
>>> c = [1, 4, 9]
>>> d = [1, 4, 9]
>>> c is d
False
```

These *equivalent*, not *identical*!
⟨ Draw pointer picture! ⟩

Q: What happens if I change a value in one of them?

A:


```
>>> c[1] = 2
>>> print(c)
[1, 2, 9]
>>> print(d)
[1, 4, 9]
```

10.11 Aliasing

We can make them identical.

```
>>> e = [1, 4, 9]
>>> f = e
>>> e is f
True
```

⟨ Draw pointer picture for e and f! ⟩

Q: Now what happens if I change `e`?

A:

```
>>> e[1] = 16
>>> print(e)
[1, 16, 9]
>>> print(f)
[1, 16, 9]
```

This can be a problem if you're not expecting it.

```
>>> a = [1, 4, 9]
>>> b = a
>>> a = [1, 2, 3]
```


Q: What is the value of `b` now?

A:

TODO: can I come up with any good functions for this?

10.12 List Arguments

Let's define some functions that remove the zeroeth element from a list!

```
>>> def delete_head(valueList):
    del t[0]
...
>>> a = [1, 4, 9]
>>> delete_head(a)
```

Q: What is the value of `a` now?

A:

⟨ Draw the pointer picture! ⟩

Q: Is `delete_head` fruitful?

A:


```
>>> a = [1, 4, 9]
>>> x = delete_head(a)
>>> x
None
```

Another!

```
>>> def other_delete_head(values):
    values = values[1:]
    return values
...
>>> a = [1, 4, 9]
>>> x = other_delete_head(a)
```

Q: What are the values of `a` and `x`?

A:

`delete_head` modifies the original list, `other_delete_head` does not.

⟨ Draw the pointer picture! ⟩

Another one!

```
>>> def another_delete_head(values):
    values = values[1:]
...
>>> a = [1, 4, 9]
>>> x = another_delete_head(a)
```

Q: What are the values of `a` and `x`?

A:

⟨ Draw the pointer picture! ⟩

Q: What does `another_delete_head` do?

A:

TODO: I can't think of any good functions for this part right now. Let students work on their projects? O:-)

11 Dictionaries

Skipped this section in class.

12 Tuples

Skipped this section in class.

13 Case Study: Data Structure Selection

Skipped this section in class.

14 Files

Skipped this section in class.

15 Classes and Objects

15.1 User-Defined Types

We have seen a few different Data Structures: strings and list. These are both *objects*.

Let's create our own type! A Pokemon type!

Def: *class* A *class* is a (user-defined) type.


```
(In Script:)
class Pokemon(object):
    '''Represents a Pokemon.'''
```

⟨ Fake-press F5 to not-really run the code. ⟩

```
(Interactive Mode)
>>> print(Pokemon)
<class '__main__.Point'>
>>> pyro = Pokemon()
>>> pyro (not print(pyro))
<__main__.Pokemon instance ...>
>>> schiggy = Pokemon()
>>> schiggy
<__main__.Pokemon instance ...>
```

Q: Why does Python print out all that nonsense?

A:

`pyro` is now an *instance* of the Pokemon class.
Def: *An* instance *of a type is a value that has that type. 5 is an instance of* ***int***
Def: *An* object *is an instance of a class.*

15.2 Attributes

We can assign blank *attributes*, which are named elements of an object.


```
>>> pyro.name = 'Flareon'
>>> pyro.number = 136
>>> pyro.types = ['Fire']
>>> pyro.hit_points = 71
>>> pyro.max_hp = 77
>>> schiggy.name = 'Squirtle'
>>> schiggy.number = 7
>>> schiggy.types = ['Water']
>>> schiggy.hit_points = 56
>>> schiggy.max_hp = 56
>>>
```

We can represent this state with an *object diagram*.
⟨ Draw figure here! ⟩

```
>>> print(pyro.hit_points)
71
>>> pyro.hit_points < pyro.max_hp
True
>>>
```

⟨ Let's define some functions on our Pokemon! ⟩

```
>>> has_full_hp(pyro)
False
>>> has_full_hp(schiggy)
True
>>> is_fainted(pyro)
False
>>> pyro.hit_points = 0
>>> is_fainted(pyro)
True
>>>
```


Q: Challenge: `def has_full_hp(pokemon):` Bonus Challenge: `is_fainted`

A:

It would be really helpful to have a function to print a `Pokemon`...

```
>>> print_pokemon(pyro)
Pokemon:  Flareon (136) HP: 0/77
>>> print_pokemon(schiggy)
Pokemon:  Squirtle (4) HP: 56/56
>>> get_types_string(pyro)
'Fire'
>>> get_types_string(schiggy)
'Water'
>>> gary = Pokemon()
>>> gary.name = 'Gyarados'
>>> gary.number = 130
>>> gary.types = ['Water', 'Flying']
>>> gary.hit_points = 83
>>> gary.max_hp = 83
>>> get_types_string(gary)
'Water/Flying'
>>>
```


Q: Challenge: `print_pokemon(pokemon)`. Bonus Challenge: `get_types_string`.

A:

15.3 Rectangles

Pokemon instead!

15.4 Instances as Return Values

TODO: I couldn't think of any functions for here.

15.5 Objects are Mutable

```
>>> print_pokemon(pyro)
Pokemon:  Flareon (136) HP: 0/77
>>> heal_up(pyro)
>>> print_pokemon(pyro)
Pokemon:  Flareon (136) HP: 77/77
>>> pyro.hit_points = 35
>>> print_pokemon(pyro)
Pokemon:  Flareon (136) HP: 35/77
>>> spray_potion(pyro)
>>> print_pokemon(pyro)
Pokemon:  Flareon (136) HP: 55/77
>>> spray_potion(pyro)
>>> print_pokemon(pyro)
Pokemon:  Flareon (136) HP: 75/77
>>> spray_potion(pyro)
>>> print_pokemon(pyro)
Pokemon:  Flareon (136) HP: 77/77
```

Q: Challenge: `heal_up(pokemon)`. Bonus challenge: `spray_potion`

A:

16 Classes and Functions

Note: This is all sectioned by my stuff since I'm not covering the Time stuff here. TODO: go through this and make it line up better with the book topics. :)

It's kind of annoying to need so mamy lines to create a Pokemon object. Let's use a new function to create them!

```
 >>> vandal = create_pokemon('Jigglypuff', number
= 39, types = ['Normal', 'Fairy'], hp = 44)
>>> print_pokemon(vandal)
Pokemon:  Jigglypuff (39) HP: 44/44
>>> schiggy = create_pokemon('Squirtle', 7,
['Water'], 56)
>>>
```

Q: Implement `create_pokemon`!

A:

16.1 Pure Functions

16.2 Modifiers

Def: *pure function* A *pure function* returns a value, but doesn't modify the parameters.

Def: *modifier* A *modifier* modifies one of the parameters, but doesn't return a value.

		Modifies Parameters?	Modifies Parameters?
		Modifies Params	Doesn't Modify
Fruitful?	Fruitful	x	Pure Function
Fruitful?	Void	Modifier	Neither (Prints?)

Q: Which of the above is `print_pokemon`?

A:

Q: Which of the above is `spray_potion`?

A:


```
>>> seventh = get_first(schiggy, pyro)
>>> print_pokemon(seventh)
Pokemon:  Squirtle (7) HP: 56/56
>>> earlier = get_first(pyro, gary)
>>> print_pokemon(earlier)
Pokemon:  Gyarados (130) HP: 83/83
>>> schiggy.hit_points = 0
>>> print_pokemon(schiggy)
Pokemon:  Squirtle (7) HP: 0/56
>>> revive(schiggy)
>>> print_pokemon(schiggy)
Pokemon:  Squirtle (7) HP: 28/56
>>> gary.hit_points = 0
>>> revive(gary)
>>> print_pokemon(gary)
Pokemon:  Gyarados (130) HP: 41/83
>>> revive(gary)
This Gyarados is not fainted.
>>>
```

Q: Code `get_first` *and* `revive`. Bonus challenge: add the guardian for `revive`.

A:

```
>>> ordered = order_pokemon(gary, pyro, schiggy)
>>> print(ordered)
[<Pokemon...>, <Pokemon...>, <Pokemon...>]
>>> for pokemon in ordered:
...       print_pokemon(pokemon)
...
Pokemon:  Squirtle (7) HP: 28/56
Pokemon:  Gyarados (130) HP: 41/83
Pokemon:  Flareon (134) HP: 77/77
>>> pokemon_list = [vandal, pyro, schiggy, gary]
>>> sort_pokemon(pokemon_list)
>>> for pokemon in pokemon_list:
...       print_pokemon(pokemon)
...
Pokemon:  Squirtle (7) HP: 28/56
Pokemon:  Jigglypuff (39) HP: 44/44
Pokemon:  Gyarados (130) HP: 41/83
Pokemon:  Flareon (134) HP: 77/77
>>>
```

Q: Code `order_pokemon`. Hard Bonus Challenge: `sort_pokemon`. (Hint: You probably need to have written the `selection_sort` project function first to get this one.)

A:


```
>>> pyro.hit_points = 0
>>> vandal.hit_points = 0
>>> fainted = get_fainted(pokemon_list)
>>> for pokemon in fainted:
...       print_pokemon(pokemon)
...
Pokemon:  Jigglypuff (39) HP: 0/44
Pokemon:  Flareon (134) HP: 0/77
>>> heal_up(vandal)
>>> injured = get_injured_but_lucid(pokemon_list)
>>> for pokemon in injured:
...       print_pokemon(pokemon)
...
Pokemon:  Squirtle (7) HP: 28/56
Pokemon:  Gyarados (130) HP: 41/83
>>>
```

Q: Code `get_fainted`. Bonus challenge: `get_injured_but_lucid`. Bonus bonus challenge: come up with a better name for that last one for me. :) Bonus bonus bonus challenge: finish `sort_pokemon`. :-P

A:

16.3 Prototyping versus Planning

17 Classes and Objects

Q: Why are we bothering to do all this with objects? Why not just use lists instead? Why don't we just use a list with three elements to represent a `Date`? (Don't let them answer yet; keep going!)


```
>>> charchar = ['Charmander', 1, ['Fire'], 23,
23]
```

17.1 Object Oriented Features

Q: What's wrong with using a list?

A:

- Names are clearer. Which of the elements above represents the month?
- We can specify what we want the attributes to be.
- Haven't seen this yet, but **methods**!

The style of programming that uses objects is called: Object-Oriented Programming. Using this makes programs easier to read, understand, and change.

OOP characteristics:

- Programs are mostly objects and functions (methods).
- Most computation consists of operations on objects.
- Objects relate to real-world concepts, as do their methods.

17.2 Printing Objects (Methods!)

Let's harness more of this and write a method!


```
(In Script):
class Pokemon(object):
    '''Represents a Pokemon.
    attributes:  name, number, types, hit_points,
max_hp'''

    def print_pokemon(pokemon):
        '''Nicely prints a pokemon.'''
        output = 'Pokemon:  '  + pokemon.name
+ ' (' + str(pokemon.number) + ') HP:
' + str(pokemon.hit_points) + '/' +
str(pokemon.max_hp)
        print(output)
```

Q: What did I actually change?

A: Moved `print_pokemon`'s definition into the class.

Q: Now `print_pokemon` is a method! How do I invoke it?

A:

⟨ Break down the syntax of the two invocation options! ⟩

Def: *subject* The *subject* of a method is the object the method is invoked on.

The idea here is that the subject takes an active role in invoking the method. Instead of: "Hey function, print this Pokemon!" we're saying: "Hey Pokemon, print yourself!"

Usually, the subject is named `self` in all methods. Let's change our script!

```
(In Script):
class Pokemon(object):
    '''Represents a Pokemon.
    attributes:  name, number, types, hit_points,
max_hp'''

    def print_pokemon(self):
        '''Nicely prints a pokemon.'''
        output = 'Pokemon:  '  + self.name
+ ' (' + str(self.number) + ') HP: ' +
str(self.hit_points) + '/' + str(self.max_hp)
        print(output)
```

Q: Would I ever need a guardian to be certain that `self` is a `Pokemon`?

A:

Q: Is the method's name very appropriate?

A: Not especially. If it's already a Pokemon, you don't need that word in the method name.

```
>>> schiggy.print()
Pokemon:  Squirtle (7) HP: 28/56
>>>
```

Let's translate some of our other functions into methods!

17.3 Another Example (Increment)

```
>>> schiggy.has_full_hp()
False
>>> pyro.is_fainted()
True
>>> schiggy.heal_up()
>>> schiggy.hit_points = 0
>>> schiggy.revive()
>>> schiggy.spray_potion()
>>> schiggy.print()
Pokemon:  Squirtle (7) HP: 48/56
>>>
```

Q: Methodize `heal_up` and `is_fainted`. Bonus challenge: do all of the others.

A:

17.4 A More Complicated Example (`is_after`)

Q: How could we "methodize" something like `get_first(poke0, poke1)`?

A: It's not really the same, since it has two subjects. Let's do something a bit different instead


```
>>> schiggy.is_after(pyro)
False
>>> pyro.is_after(gary)
True
>>> pokemon_list = [schiggy, vandal, pyro]
>>> gary.heal_up()
>>> gary.more_hp_than_all(pokemon_list)
True
>>> pyro.more_hp_than_all([schiggy, vandal, gary])
False
>>>
```

Q: Methodize it! Header: `def is_after(self, other)`
Bonus Challenge: `more_hp_than_all`

A:

We can change the syntax for creating new objects!

17.5 The init Method

```
>>> mankey = Pokemon()
>>> mankey.initialize("Mankey", 56, ['Fighting'],
51)
>>> mankey.print()
Pokemon:  Mankey (56) HP: 51/51
>>> bulby = Pokemon()
>>> bulby.initialize()
>>> bulby.print()
Pokemon:  Bulbasaur (1) HP: 1/1
>>>
```

This is a bit different than `create_pokemon`... but I'm getting somewhere, don't worry!

Q: Implement it! Header:

```
def initialize(self, number = 1, name =
'Bulbasaur', types = ['Grass'], hp = 1):
```

A:

Then I renamed mine to have this funny name:

```
>>> bulby = Pokemon()
>>> bulby.__init__(hp = 10)
>>> bulby.print()
Pokemon:  Bulbasaur (1) HP: 10/10
>>>
```

Now it's time for something crazy!

```
>>> bulby = Pokemon(hp = 10)
>>> bulby.print()
Pokemon:  Bulbasaur (1) HP: 10/10
>>>
```

Q: How did that happen?

A:


```
>>> mankey = Pokemon("Mankey", 56, ['Fighting'],
51)
>>> mankey.print()
Pokemon:  Mankey (56) HP: 51/51
>>>
```

Try it out!

Q: What is weird about `__init__`?

A:

Q: So what happens here?

A: Python uses the `__init__` method (called a *constructor* you write to define the attributes. Two things happen after you create an object:

- First, the object is created, then
- The `__init__` method is called with whatever parameters were given to the creator.

Let's create another class!


```
>>> mw_latitude = GeographicCoordinate('N', 44,
16, 13.8)
>>> mw_longitude = GeographicCoordinate('W', 71,
18, 11.7)
>>> print_coordinate(mw_latitude)
44°16'13.8"N
>>> print_coordinate(mw_longitude)
71°18' 11.7"W
>>> x = GeographicCoordinate('S', 110, 11, 3.2)
Error!
>>> x = GeographicCoordinate('X', 1, 1, 1)
Error!
>>>
```


Q: Implement `GeographicCoordinate.__init__`. Important: guardians are needed! Bonus challenge: implement `print_coordinate`

A:

Def: *special method* A Python method that uses the four under-

scores is known as a *special method*. We'll see more soon! Special methods can be invoked without explicitly typing out the call.

Q: Are there more special methods we can use?

A: Absolutely!

17.6 The __str__ Method

```
>>> abra = Pokemon('Abra', 63, ['Psychic'], 22)
>>> print(abra)
Pokemon:  Abra (63) HP: 22/22
>>> print(Pokemon())
Pokemon:  Bulbasaur (1) HP: 1/1
>>>
```

Q: Wow! How did that happen?

A: Another special method: `__str__`!

Q: Write it!

A:

Q: When is this called?

A:

Q: Why is this method better than the `print_pokemon` function?

A:

Q: When creating a new class, which should be the first methods you define?

A:

Q: Why?

A:

I created some new classes with __init__ and __str__ methods.

```
>>> mw_latitude = GeographicCoordinate('N', 44,
16, 13.8)
>>> mw_longitude = GeographicCoordinate('W', 71,
18, 11.7)
>>> print(mw_latitude)
44°16'13.8"N
>>> print(mw_longitude)
71°18'11.7"W
>>> mw_latiude.is_latitude()
True
>>> mw_latitude.is_longitude()
False
```


Q: Challenge: Implement `__str__`. Bonus challenge: implement `is_latitude` and `is_longitude`.

A:

```
>>> mw_location = Location(mw_latitude,
mw_longitude)
>>> print(mw_location)
(44°16’13.8"N, 71°18’ 11.7"W)
>>> mw_backwards = Location(mw_longitude,
mw_latitude)
This is not a legitimate location!  Longitude and
latitude are incorrect!
>>>
```


Q: Implement `Location` and the two basic methods (`__init__` and `__str__`. Bonus challenge: build the guardian in to the constructor to make sure they're in the right places!

A:


```
>>> washington = Mountain('Mount Washington',
1916.6, mw_location)
>>> print(washington)
Mount Washington (1916.6m tall @(44°16'13.8"N
71°18' 11.7"W))
>>> nearby = Mountain('Plymouth Mountain',
670, Location(GeographicCoordinate('N', 43,
42, 32.04), GeographicCoordinate('W', 71, 43,
24.96)))
>>> print(nearby)
Plymouth Mountain (670m tall @(43°42' 32.04"N
71°43'24.96"W))
>>> pemi = River('Pemigewasset River', 104.6,
Location(GeographicCoordinate('N', 43, 26, 12),
GeographicCoordinate('W', 71, 38, 55)))
>>> print(pemi)
The Pemigewasset River is 104.6km long.  Its
mouth is at (43°26'12"N, 71°38'55"W).
>>>
```


Q: Code the `Mountain` class, with `__init__` and `__str__` methods. Bonus challenge: do the same for `River`.

A:

```
>>> washington.taller_than(nearby)
True
>>> nearby.taller_than(washington)
False
>>> washington.taller_than(washington)
False
```

Q: Implement `taller_than`. Bonus: do it in one line.

A:


```
>>> perfect_spring_day = WeatherReport("Plymouth",
63, 49, "Partly Cloudy")
>>> print(perfect_spring_day)
...
>>> rainy_munich_day = WeatherReport("Muenchen",
13, 18, "Scattered Showers", "Celsius")
>>> print(rainy_munich_day)
...
>>>
```


Q: Code `WeatherReport` (without celsius). Bonus challenge: do the celsius $\leftrightarrow$ fahrenheit conversion. (Hint: I wrote a separate function to go back and forth.)

A:

18 Inheritance

To skip to new stuff not covered in the textbook, jump to Section 18.7.

18.1 Card Objects

Q: Let's play cards! What do we need?

A:

Q: What are our attributes going to be?

A:

Q: Hard: What types will we use for our attributes?

A:

What? How does that work? We're going to encode the different suits:

- 3 means Spades
- 2 means Hearts
- 1 means Diamonds
- 0 means Clubs

Q: How could we encode the card values? (Ace, King, ..., 2)

A:

Q:
```
>>> spade_five = Card(3, 5)
```
Write the class and the constructor!
Bonus Challenge:
```
>>> club_two = Card()
```
Write it with default parameters to Clubs and 2.

A:

Q:

```
>>> spade_five = Card(3, 5)
>>> print(spade_five)
5 of Spades
>>> club_two = Card()
>>> print(club_two)
2 of Clubs
>>> print(Card(2, 12))
Queen of Hearts
```

Write `__str__`. Bonus challenge: write it so the Jack, Queen, King, and Ace are printed nicely like this.

A:

18.2 Class Attributes

So far, things are a bit inelegant. Let's do some refactoring to improve things!

```
(Script)
class Card(object):
    '''Models a playing card.  Attributes:  suit
and rank.'''

    suit_names = ['Clubs', 'Diamonds', 'Hearts',
'Spades']
    rank_names = [None, 'Ace', '2', '3', '4',
'5', '6', '7', '8', '9', '10', 'Jack', 'Queen',
'King']
```

We can access these in the following way:

```
>>> print(Card.suit_names[2])
Hearts
>>> print(Card.rank_names[12], 'of',
Card.suit_names[0])
Queen of Clubs
```

These variables inside a class but not inside any methods are known as *class attributes*. Unlike *instance attributes*, these are not associated with a specific instance of the class. They are instead used by the class "at large".

Q: Those big conditionals are kind of ugly. Refactor `__str__` to use these lists. (Hint: the result will be much shorter!) Bonus Challenge: do it using print formatting.

A:

Q: If I look at a line of code: `XXXX.YYYY`, how do I know whether `YYYY` is a class or instance attribute?

A:

Q: How should you be able to tell just by looking at the name of `XXXX`?

A:

Draw this diagram showing the class and the class attributes:
TODO: This wasn't compiling, so I commented it out.

⟨ ... and this diagram showing an object and its instance attributes:
TODO: commented out the "diagram" because it wasn't compiling.
⟩

18.3 Comparing Cards

Warning: I think this part of the book is wrong. In Python 3, the `__cmp__` method doesn't work the same way as it does in Python 2.

There are another pair of special methods in Python (3):

- `__lt__`: "less than"
- `__eq__`: "equals"

If I implement them, then I can start using the boolean comparison operators. (`<`, `==`, and others!)

Q: What should `__eq__(self, other)` return?

A:

Q: Implement `__eq__` for the `Card` class.

A:

Q: Implement `__lt__` for `Card`. Cards should first compare their rank, then break ties based on the suit.
Bonus challenge: write it in one line. (Hint: one long boolean statement)
Super Bonus challenge: Write it so that the Ace is the highest card in the set.

A:

Now let's play with these methods.

```
>>> six = Card(1, 6)
>>> queen = Card(2, 12)
>>> queen < six
False
>>> six == queen
False
>>> six < queen
True
>>> clubs_queen = Card(0, 12)
>>> clubs_queen < queen
True
>>> queen < clubs_queen
False
>>> queen > six
ERROR!
```


Q: Why did that last one error?

A:

There are a bunch more comparison methods to implement:

- `__gt__`: $>$
- `__ge__`: $\geq$
- `__le__`: $\leq$
- `__ne__`: $\neq$

Thankfully, there's a good way around this one so we don't have to implement all these methods: we can use the `total_ordering` decorator:

```
(Script)
from functools import total_ordering

@total_ordering
class Card(object):
    ...
```

Now we can do all the comparing:

```
>>> six = Card(1, 6)
>>> queen = Card(2, 12)
>>> clubs_queen = Card(0, 12)
>>> queen >= clubs_queen
True
>>> six != queen
True
```

18.4 Decks

Now let's make a new object, a deck of cards!


```
class Deck(object):
    '''Represents a deck of cards.
    attributes:  cards'''

    def __init__(self):
        '''Creates a new Deck object with all 52
cards.'''
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

Wow! We just wrote a nested loop to create an object! Cool!

18.5 Printing the Deck

Q: What method should we write next?

A:

Q: How do we want this to work?

A:

Q: Write it! Bonus Challenge: write it so that it numbers the cards, like this:

```
0:  2 of Clubs
1:  3 of Clubs
...
51:  Ace of Spades
```

A:

18.6 Add, remove, shuffle, and sort

Q: What operation might we want to often do from a deck of cards?

A:

Q: Great! Write a `deal()` method that removes the next card and returns it. Here's what should happen:

```
>>> card = deck.deal()
>>> print(card)
King of Spades
>>> print(deck.deal())
Queen of Spades
```

Hint: There's a list method that does this in one line.

A:

Q: Is `deal()` a pure function or a modifier?

A:

```
>>> hand = []
>>> for i in range(5):
...     hand.append(deck.deal())
...
>>> print(hand)
[<Card object ...>, ...blahblah...]
>>> for i in range(5):
...     print(hand[i])
...
Jack of Spades
10 of Spades
9 of Spades
8 of Spades
7 of Spades
```


Q: Oooh! A flush! Am I super lucky?

A:

Q: What should I do with the deck before I start dealing?

A:

```
(shuffle example)
>>> numbers = [1, 2, 3, 4, 5, 6]
>>> import random
>>> random.shuffle(numbers)
>> print(numbers)
[some random permutation...]
```

Q: Let's add a `shuffle()` method to the `Deck` class that shuffles the cards. Add it!

A:


```
>>> deck = Deck()
>>> deck.shuffle()
>>> hand = []
>>> for i in range(5):
...     hand.append(deck.deal())
...
>>> for i in range(5):
...     print(hand[i])
...
5 random cards
```

Q: How many times should we shuffle the deck?

A:

Q: It's a bit annoying to write a for loop everytime we need to deal out a hand. What should we do instead?

A:

Q: Do it! `deal_hand`

A:

Q: Okay, what's annoying about this?

A:

Q: Luckily, there's a nice way to get it to print. We could turn each hand into an object, then use that object's `__str__` method to print the cards. A hand is actually like a small version of a type we've already created. Which one?

A:

I've modified `deal_hand` to return a `Deck`.

```
>>> deck = Deck()
>>> deck.shuffle()
>>> hand = deck.deal_hand()
>>> print(hand)
...  5 random cards...
```


Q: Write `deal_hand` so that the above code works. Hint: I modified my `Deck.__init__` method by changing the first two lines:

```
def __init__(self, cards = None):
    '''Constructor.'''
    self.cards = cards
    if self.cards == None:
```

A:

Q:

We might also want a way to put cards back into the deck:

```
>>> deck = Deck()
>>> len(deck.cards)
52
>>> deck.shuffle()
>>> card = hand.deal()
>>> len(deck.cards)
51
>>> deck.add_card(card)
>>> len(deck.cards)
52
```

Write `add_card`!

Bonus Challenge: write `Deck.size()`

A:

18.7 Inheritance

Let's take a slight detour from things in the book...

I created a new class:

```
>>> fuji = Volcano('Mount Fuji', 3,776.24,
Location(GeographicCoordinate('N', 35, 21, 29),
GeographicCoordinate('E', 138, 42, 52)))
>>> print(fuji)
Mount Fuji (3776.24m tall @ (35°21' 29"N,
138°43'52"E))
>>> fuji.taller_than(washington)
True
>>> fuji.is_active()
True
>>
```


Q: Do we have to copy/paste a lot of code from `Mountain`?

A: No! We can use *inheritance*!

```
class Volcano(Mountain):
    ''' Models a volcano.'''

    def __init__(self, name, height_in_m, latitude,
longitude, active=False):
        '''Class constructor.'''
        self.name = name
        self.height = height_in_m #in meters
        self.location
        self.active = active
```

Q: Try that out. Without doing anything else, which `Volcano` methods work? Which don't?

A: `__str__` and `taller_than` work, but `is_active` does not.

Q: Why is that?

A: `Volcano` inherits all of the attributes and methods from `Mountain`!

Q: Implement `is_active`

A:

Q: Is `fuji` a `Mountain`? Is `washington` a `Volcano`? How can we be sure in Python?

A:

```
>>> isinstance(fuji, Mountain)
True
>>> isinstance(washington, Volcano)
False
>>>
```

```
>>> fuji.add_eruption(1708)
>>> fuji.add_eruption(864)
>>> fuji.last_eruption()
1707
>>> kilauea = Volcano('Kilauea', 1247,
Location(GeographicCoordinate('N', 19, 25, 16),
GeographicCoordinate('W', 155, 17, 12)))
>>> kilauea.add_eruption(1968)
>>> kilauea.add_eruption(1974)
>>> kilauea.add_eruption(2018)
>>> kilauea.last_eruption()
2018
>>> fuji.more_recent_eruption_than(kilauea)
False
```

Q: What do I need to modify before writing the methods.

A:

Q: Modify `__init__`, then implement `add_eruption` and `last_eruption`. Bonus challenge: `more_recent_eruption_than`.

A:

Q: Inheritance is useful for removing repeated code. Is there any other code that is repeated between `Volcano` and `Mountain`?

A: Yes! Inside the constructors!

Q: Is there a way to remove that repetition?

A: Yes!


```
class Volcano(Mountain):
    '''Models a volcano.'''

    def __init__(self, name, height_in_m, latitude,
longitude, active=False):
        '''Class constructor.'''
        super().__init__(name, height_in_m,
location)
        self.active = active
        self.eruptions = []
```

Q: What does `super()` do?

A: It grabs the super class. Then we can explicity call those methods!

Q: Can we use `super()` to make a better `__str__` for `Volcano`?

A: Yes!

```
(Note, this won't work on mine without breaking
it for the future)
>>> mountains = [fuji, washington, nearby,
kilauea]
>>> for mountain in mountains:
...     print(mountain)
...
Mount Fuji (3776.24m tall @ (35°21' 29"N,
138°43'52"E)) is an active volcano
Mount Washington (1916.6m tall @(44°16'13.8"N
71°18' 11.7"W))
Plymouth Mountain (670m tall @(43°42' 32.04"N
71°43'24.96"W))
Kilauea (1247m tall @ (19°25' 16"N, 155°17' 12"))
is an active volcano
>>>
```

Q: How does Python know which class's version of `__str__` to execute?

A: Python checks the objects type when it's running, and sees if there's a method of that name in that class. If not, it goes up to the super class and checks that, etc. This is called `polymorphism`.

Q: How far up does this checking go?

A: Until:

- It finds a method with the correct name, or
- It goes all the way up to `object` and doesn't find anything!

Q: How high does this chain go?

A: There's no limit!

Q: How could we add child classes of `Volcano` so that there's no `active` field?

A:

Q: Implement both of them! Then remove `active` as a field from the Volcano class.

A:

Q: Now that we've remove the `active` field, which method should we remove from the `Volcano` class?

A: `is_active`

Q: Should we have `is_active` be a method in the subclasses?

A: Sure! And now they're very easy to write!

Q: Implement them both!

A:

Now back to the stuff from the book...

Hands aren't exactly the same thing as Decks, though. There are some things we'd like to do differently. For example, it would be great to create hands like this:

```
>>> deck = Deck()
>>> hand = Hand(deck, 5)
>>> print(hand)
...  five random cards...
>>> another_hand = Hand(deck, 5)
>>> print(another_hand)
...  five different random cards...
```


Q: Which is different from `Deck` and which could be the same?

- `__init__`
- `__str__`

A:

Q: Do I need a new `Hand` class?

A:

Let's use *inheritance*! This is where a class will borrow some implementation from another class. Here's the code: (first line is super important! Deck instead of object!)

```
class Hand(Deck):
    '''A hand of playing cards.
    attributes:  cards'''
```

Now, `Hand` automatically uses `Deck`'s methods *unless* we specify a new implementation here.

Q: Do we need to implement a constructor?

A:

Q: Implement it!

A:

Q: Do we need to implement `Hand.__str__`?

A:

Let's try it out! Test the code and see that it works! :)

18.8 Class Diagrams

TODO: show them the class diagram including: Hand, Deck, and Card.

18.9 Additional `Hand` Operations

Q: Is there other functionality we might want to add to the `Hand` class we could encapsulate into a method?

A:

Q: Let's try to do the draw a card method. What should the interface be?

A:

Q: Code it up! Bonus Challenge: implement one of the others.

A:

